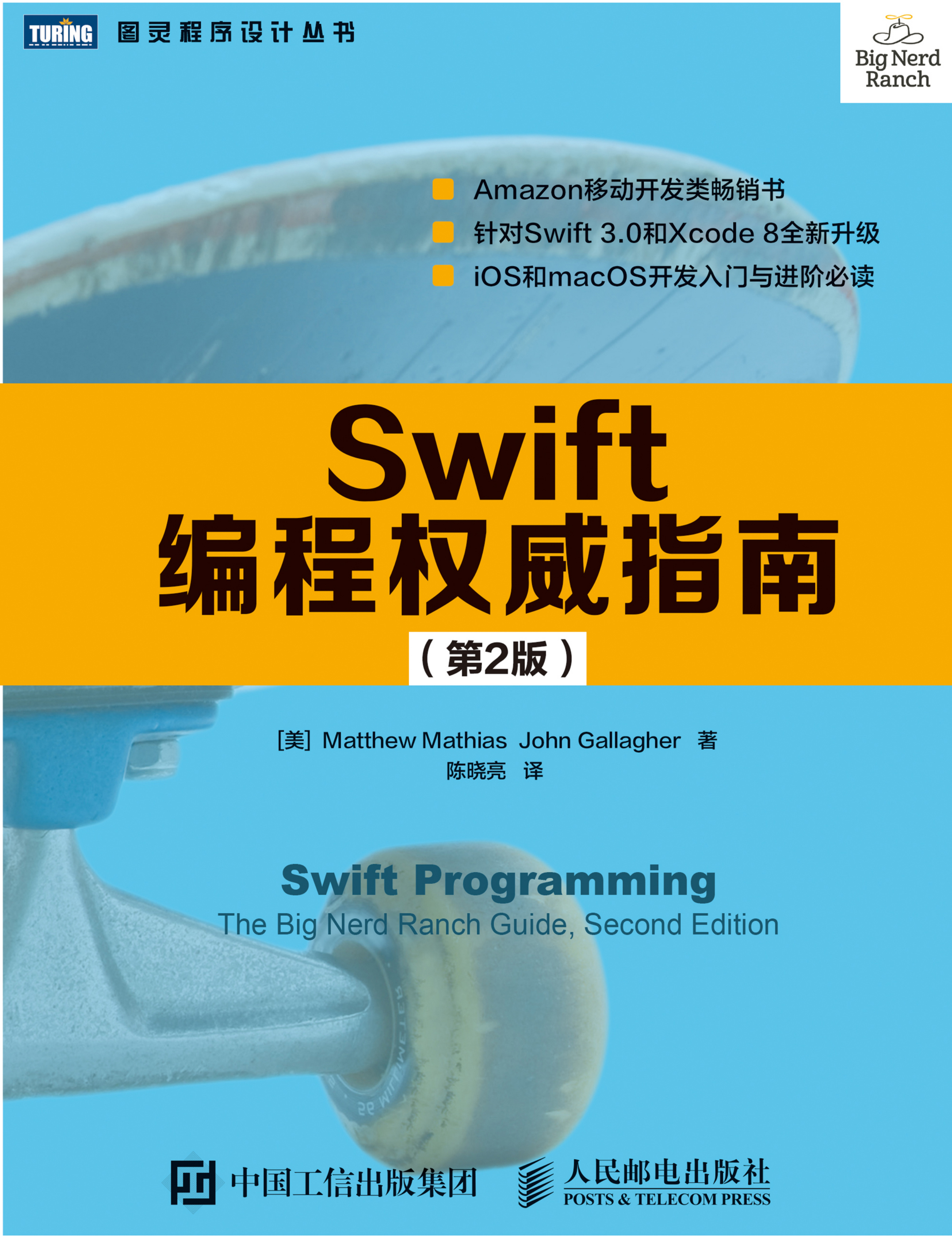


TURING 图灵程序设计丛书

Big Nerd Ranch

- Amazon移动开发类畅销书
- 针对Swift 3.0和Xcode 8全新升级
- iOS和macOS开发入门与进阶必读

Swift 编程权威指南

（第2版）

[美] Matthew Mathias John Gallagher 著
陈晓亮 译

Swift Programming

The Big Nerd Ranch Guide, Second Edition

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Matthew Mathias

Big Nerd Ranch教学主管、iOS讲师。社会学博士，曾在高校任教。目前致力于通过Big Nerd Ranch为广大编程爱好者提供最好的编程教材和培训。工作之余，Matt喜欢骑车、看漫画、打游戏等。

John Gallagher

Big Nerd Ranch软件工程师、讲师。除了为客户开发App，他的大部分职业生涯都在与非常小的嵌入式系统和非常大的超级电脑打交道。他喜欢学习新的编程语言，并且寻找将其组合使用的方法。工作之余，John的大部分空闲时间都与妻子和三个女儿在一起。

TURING 图灵程序设计丛书

Swift 编程权威指南

（第2版）

[美] Matthew Mathias John Gallagher 著
陈晓亮 译

Swift Programming

The Big Nerd Ranch Guide, Second Edition

人民邮电出版社
北 京

图书在版编目（C I P）数据

Swift编程权威指南 / （美）马修·马赛厄斯（Matthew Mathias），（美）约翰·加拉格尔（John Gallagher）著；陈晓亮译. -- 2版. -- 北京：人民邮电出版社，2017.6
（图灵程序设计丛书）
ISBN 978-7-115-45746-2

Ⅰ. ①S… Ⅱ. ①马… ②约… ③陈… Ⅲ. ①程序语言—程序设计—指南 Ⅳ. ①TP312-62

中国版本图书馆CIP数据核字(2017)第112833号

内 容 提 要

Big Nerd Ranch 是美国一家专业的移动开发技术培训机构，本书是其培训教材。书中系统讲解了在 iOS 和 macOS 平台上，使用苹果的 Swift 语言开发 iPhone、iPad 和 Mac 应用的基本概念和编程技巧。主要围绕使用 Swift 语言进行 iOS 和 macOS 开发，结合大量代码示例，教会读者利用高级 iOS 和 macOS 特性开发真实的应用。

本书读者对象为 iOS 和 macOS 平台移动开发人员。

◆ 著　　[美] Matthew Mathias　John Gallagher
译　　陈晓亮
责任编辑　杨　琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000　1/16
印张：24.25
字数：587千字　　2017年 6 月第 1 版
印数：1 – 4 000册　　2017年 6 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-2563号

定价：89.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

献给我聪明、坚强、善良的妻子。也献给我的家人，是他们给了我美好的生活。

——Matthew Mathias

献给我的妻子兼最好的朋友；你真是“太棒了”。也献给每天为我带来欢乐的女儿们。

——John Gallagher

前　　言

学习 Swift

苹果公司的全球开发者大会（World Wide Developers Conference，WWDC）是其开发者社区一年一度的盛事。虽然WWDC每年都是大事件，但在2014年尤为特殊：苹果为iOS和OS X（现在称为macOS）应用开发推出了一门全新的语言Swift。

作为一门新语言，Swift对macOS和iOS开发者意味着相当巨大的转变。对于资深iOS开发者来说，他们得学些新东西了；而对于新手开发者，也还没有成熟的社区来获取久经考验的答案和解决问题的模式。很自然，这个转变会造成一些不确定性。

但是对macOS和iOS开发者来说，这也是个激动人心的时刻。我们可以从一门新语言中学到很多东西，对Swift而言更是如此。这门语言从2014年夏天发布以来已经进化了不少，而且还在持续进化。

我们都处于这门语言发展的最前沿。随着Swift新特性的增加，使用者可以一起合作来摸索出最佳实践。你可以直接对这场讨论发表意见，在本书的指导下进行开发也将开启你对Swift社区的贡献之旅。

为什么选择 Swift

如果有过苹果平台上的Objective-C开发经验，你可能会想，苹果为什么还要发布一门新语言呢？毕竟这些年来，开发者产出了不少高质量的OS X和iOS应用。苹果其实考虑了几件事情。

首先，Objective-C是比较老的语言。虽然老不一定会出问题，但是确实造成了一些困难。Objective-C的语法在20世纪90年代脚本语言兴起之前就固化了，这些脚本语言普及了更精简、更优雅的语法（比如JavaScript、Python、PHP、Ruby等）。这让大部分开发者在起步的时候觉得Objective-C的语法很奇怪，并可能造成开发者产出的降低。此外，作为一门比较老的语言，Objective-C也缺失了现代语言的开发者们正在享受的很多高级特性。

其次，Swift的目标是安全。虽然Objective-C的目标并非不安全，但是自从Objective-C在20世纪80年代发布以来，情况发生了很多变化。举个例子，Swift编译器致力于尽量减少未定义行为，从而减少开发者对在应用运行时出错的代码进行调试的时间成本。

Swift的另一个目标是成为C系语言（C、C++和Objective-C）的替代品。这意味着Swift必须有足够快的运行速度。的确，Swift在多数情况下的性能和这些语言是差不多的。

Swift用一种干净、现代的语法提供了安全性和性能。这门语言表达力强，可以让开发者写出很自然的代码。这让Swift代码写起来很舒服，读起来很容易，适合用于大项目中的协作。

最后，苹果想让Swift成为通用编程语言。这一点从2015年12月Swift开源就能反映出来。把语言开源不仅能吸引开发者进来帮助语言进化，还能让开发者更容易地把这门语言移植到macOS和iOS以外的系统上。苹果希望开发者能用Swift开发多种移动和桌面平台上的应用，以及开发后端Web应用。Swift意在成为一门主流编程语言，为在多种平台上开发多种应用提供最好的解决方案。

Objective-C 前景如何

那么，苹果所有平台的上一代*通用语言*Objective-C的前途如何？开发者还需要懂这门语言吗？就专业的macOS和iOS开发者来说，我们认为答案是明确的，“需要”。苹果广泛使用的Cocoa和UIKit框架是用Objective-C写的，学会这门语言会使调试变得更加简单。实际上，苹果已经让同一工程中Swift和Objective-C混编变得容易——混编有时候还是更优的选择。macOS或者iOS开发者势必会接触Objective-C，所以应该熟悉这门语言。

那么，学习Swift或者开发macOS或iOS应用需要有Objective-C的基础吗？完全不需要。Swift和Objective-C并存，也可以互操作，但是Swift自成一体。就算不会Objective-C，也不会妨碍你学习Swift。（全书只有第28章会直接使用Objective-C，不过即使读到那一章时，不懂这门语言也不要紧。）

本书读者对象

本书是写给从初学者到平台专家的各层次macOS和iOS开发者看的。针对刚接触软件开发的读者，我们会突出并实现Swift和通用编程方法的最佳实践。我们的策略就是在教你学习Swift的同时帮你打下编程基础。至于有经验的开发者，我们相信本书能帮你快速入门所在平台的新语言。具备一定的开发经验当然很好，但是没有这样的经验一样可以阅读本书。

本书使用了大量的示例，以便读者在将来的开发过程中参考。这些示例不会着眼于抽象的概念和理论，而是更倾向于实用性。我们喜欢用实际的例子来剖析艰深的概念，从而让大家了解最佳实践，让代码更有趣、更可读、更易维护。

本书内容

本书分为六个部分，每个部分都会完成一组特定的目标，且每个目标都以彼此为基础。读完本书后，你将不再是一个初学者，而是会完成高级开发者的Swift知识构建。

- **第一部分　起步**

这部分着眼于写Swift代码所需的工具，并且会介绍Swift的语法。

- **第二部分　基础知识**

这部分介绍Swift开发者每天都会用到的基本数据类型，还涵盖了Swift用来控制代码执行顺序的控制流特性。

- **第三部分　容器和函数**

在应用中经常需要收集相关数据，一旦收集完成，还需要对这些数据进行操作。这部分讲解Swift提供的来帮你完成这些任务的容器和函数。

- **第四部分　枚举、结构体和类**

这部分阐述了如何在开发过程中为数据建模。我们会讨论Swift的枚举、结构体和类之间的差异，并就每种类型适合何时使用给出建议。

- **第五部分　Swift高级编程**

作为一门现代语言，Swift提供了更加高级的特性，能让你写出优雅、可读、高效的代码。这部分讨论如何使用Swift的这些元素写出区别于普通Swift开发者的地道代码。

- **第六部分　事件驱动的应用**

这部分将指引你写出第一个macOS和iOS应用。对于之前写过macOS和iOS应用的读者，我们将在其结尾讨论如何在Objective-C和Swift之间互操作。

如何使用本书

编程有时候很难，本书尝试让编程变得更容易。要发掘本书的最大价值，我们建议你遵循以下步骤。

- 精读本书。必须精读！别只是在晚上睡觉前随便翻翻。
- 阅读过程中把示例代码敲出来。肌肉记忆是学习的一部分。如果不需要太多思考，手指就知道该怎么动、该敲什么键，那么你就已经迈上了成为高效开发者之路。
- 犯错误！根据我们的经验，学习事物运作原理的最好办法是先搞清楚什么情况下会出问题。可以先把示例代码弄坏，再让代码跑起来。
- 根据自己的想象力进行实验。无论是微调本书中的代码，还是尝试自己的方向，越早用Swift解决自己的问题，就能越早成为更好的开发者。
- 完成我们提供的挑战练习。就像刚才说的，尽早用Swift解决问题很重要，这能让你像开发者一样思考。

有经验的开发者可以不阅读本书的前几章。有些开发者很熟悉第一部分和第二部分中的内容。

警告：在第二部分中，不要跳过第8章，因为那是Swift的核心，而且用多种方式定义了这门语言的独特之处。

后面的第9章、第10章、第12章、第14章和第15章，对于熟练的开发者来说可能不是什么新

知识，但是我们认为Swift实现这些类型的方式十分独特，读者最起码应该快速浏览一下这几章。

最后，记住学习新知识很花时间。在可以心无旁骛的时候专门拿出一段时间来好好阅读本书，你能从书里得到更多的收获。

挑战练习

很多章的结尾都有练习，你应该独立完成。这是挑战自己的好机会。在练习中独立解决问题能真正完成深入学习。

深入学习

我们还在很多章的结尾加上了“深入学习”一节。这部分内容解答那些好奇的读者在阅读相应一章时可能提出的问题。有时候，我们会讨论语言的某个特性的底层机制，或者探索某个跟那一章的核心内容不大相关的编程概念。

排版约定

在阅读本书过程中，你会写很多代码。为了让这个过程更容易，我们有一些约定来定义哪些代码是现有的，哪些应该添加进来，哪些应该删除。比如下面这个函数实现，你需要删除`print("Hello")`并添加`print("Goodbye")`。`func talkToMe() {`那一行和右花括号`}`是已有的代码。这样能帮你定位代码变动发生的位置。

```
func talkToMe() {
    print("Hello")
    print("Goodbye")
}
```

必要的硬件和软件

要构建并运行本书中的应用，你需要一台运行macOS El Capitan（10.11.4）或更新系统的Mac，还需要安装苹果的集成开发环境（integrated development environment，IDE）Xcode（App Store上有）。Xcode包含了Swift编译器以及其他在阅读本书过程中所需的开发工具。

Swift还在快速发展之中。本书是针对Swift 3.0和Xcode 8.0编写的。如果你用老版本的Xcode，书中的很多例子就运行不了。如果用更新版本的Xcode，也可能存在语言本身发生了变化而导致有些例子运行失败的情况。

在本书付印过程中，Xcode 8.1 Beta也可以下载了。本书中的示例代码在最新beta版上能正确运行。如果未来新版Xcode真的会产生问题也别担心：即使在语法上或命名上有些区别，你学到的大部分知识对未来新版的Swift也都适用。你也可以访问我们的论坛http://forums.bignerdranch.com寻求帮助。

开始之前

我们希望向你展示，为苹果生态系统制作应用是多么有意思。写代码，有时候让人感到极度挫败，有时候又让人欣喜。解决问题拥有魔力，能让人兴奋，更别提制作出一款能帮助用户并给他们带来欢乐的应用所带来的愉悦心情了。

取得进步的最好办法是练习。如果你想成为开发者，那就开始吧！就算发现自己不是很擅长编程，那又怎么样？继续练习，我们相信有一天你会让自己大吃一惊。路就在脚下，前进吧！

致　　谢

我们在编写本书的过程中得到了大量帮助。如果没有这些帮助，本书将远不如现在这么好，甚至根本无法完成。我们应该表示感谢。

首先，我们要对Big Nerd Ranch的同事说声谢谢。感谢Aaron Hillegass为我们提供了编写本书的机会。学习和教授Swift编程是一件令人欣喜的事情。Big Nerd Ranch为我们提供了完成这个项目的时间和空间。我们希望本书没有辜负所获得的信任与支持。

我们还要感谢Big Nerd Ranch的Cocoa团队的同事。你们的认真教学暴露出本书的很多问题，你们周到的建议让我们的工作方法得到了很多改进。还有很多本身不是讲师的同事帮我们审查了材料和工作方法，并提供了我们永远都想不到的无数建议。有你们这样的同事真好。谢谢Pouria Almassi、Matt Bezark、Nate Chandler、Step Christopher、Kynerd Coleman、Matthew Compton、Joseph Dixon、Robert Edwards、Sean Farrell、Brian Hardy、Florian Harr、Tom Harrington、David House、Bolot Kerimbaev、Christian Keur、JJ Manton、Bill Monk、Chris Morris、Adam Preble、Scott Ritchie、Jeremy Sherman、Steve Sparks、Rod Strougo、TJ Usiyan、Zach Waldowski、Thomas Ward和Mike Zornek。

我们在运营、市场和销售部门的同事也提供了很大帮助。如果没有他们的工作，我们永远都不可能把课程排出来。谢谢Shannon Coburn、Nicole Rej、Heather Sharpe、Tasha Schroeder、Jade Hill、Nicola Smith、Mat Jackson、Chris Kirksey和Jon Malmgren，谢谢你们的辛勤工作。我们做不了你们的工作。

接下来，我们想感谢很多和我们一起为本书付出努力的有才华的人。

我们的编辑Elizabeth Holaday帮助我们完善了这本书，去芜存菁。

我们的编审Simone Payment和Anna Bentley发现并修复了很多错误，让我们看起来比实际上更聪明。

Ellie Volckhausen设计了本书的封面， 那块滑板看起来很漂亮。

Chris Loper设计和制作了纸质版，以及EPUB和Kindle版本。

最后，感谢我们的学生。我们和你们一起学习，更为你们而学习。教学是我们做过的最棒的事情，和你们一起学习令我们愉悦。我们希望本书的质量能对得起你们的热情和决心。

目　录

Part 1

第一部分

起　　步

这部分介绍编写Swift代码所需的工具链，包括Swift开发者的主要开发工具Xcode，并且使用playground来提供试运行代码的轻量环境。这几章还会帮你熟悉一些Swift最基本的概念，比如常量和变量，以便为学习本书的后续部分和深入理解这门语言打好基础。

第 1 章 起步

本章将介绍如何设置环境，并简要了解iOS和macOS开发者日常使用的一些工具。此外，你还会自己动手写些代码以更好地了解Swift和Xcode。

1.1 Xcode 起步

如果还没有安装Xcode，可以从App Store下载并安装。确保下载Xcode 8或者更新的版本。

安装完成后，启动Xcode。欢迎画面会给出一些选项，包括Get started with a playground和Create a new Xcode project（如图1-1所示）。

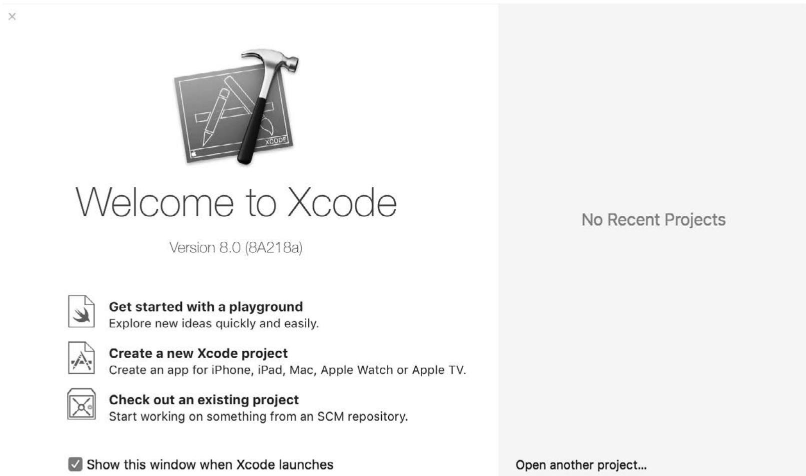


图1-1　Xcode欢迎画面

playground是随着Xcode 6发布的，能为你提供交互环境用来快速开发及运行Swift代码，已经成为了一个实用的原型工具。playground不需要编译运行整个工程，而是随时运行Swift代码，所以非常适合在轻量环境中测试和试验Swift语言。在阅读本书的过程中，你会经常用到playground，从所写的Swift代码快速得到结果。

除了playground之外，后面几章还要创建命令行工具。为什么只用playground不够呢？因为那

样会错过Xcode的很多特性，从而不能充分了解这个IDE。你之后会在Xcode上花费很多时间，所以最好尽早适应。

在欢迎画面选择Get started with a playground。

接下来，把playground命名为MyPlayground。选择平台时（iOS、macOS或tvOS），即使你是iOS开发者也请选择macOS（如图1-2所示）。我们即将用到的Swift特性对两个平台是通用的。点击Next。

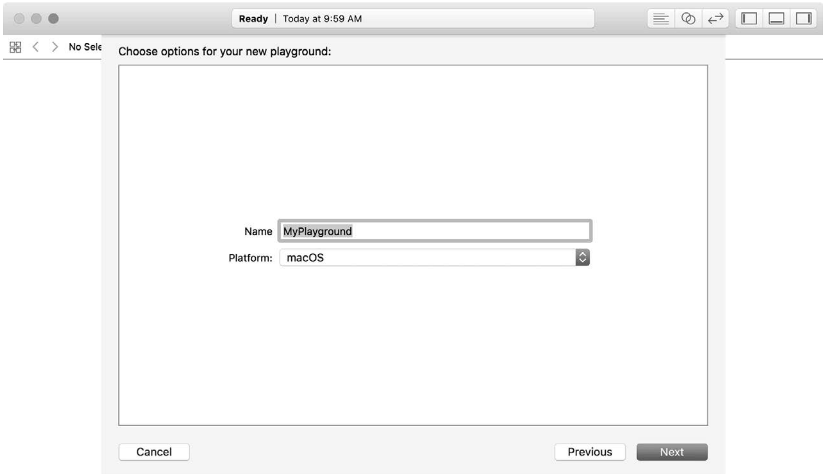


图1-2　命名playground

最后，Xcode会提示保存playground。在阅读本书的过程中，最好把所有的代码放到一个目录中。选择一个合适的路径并点击Create（如图1-3所示）。

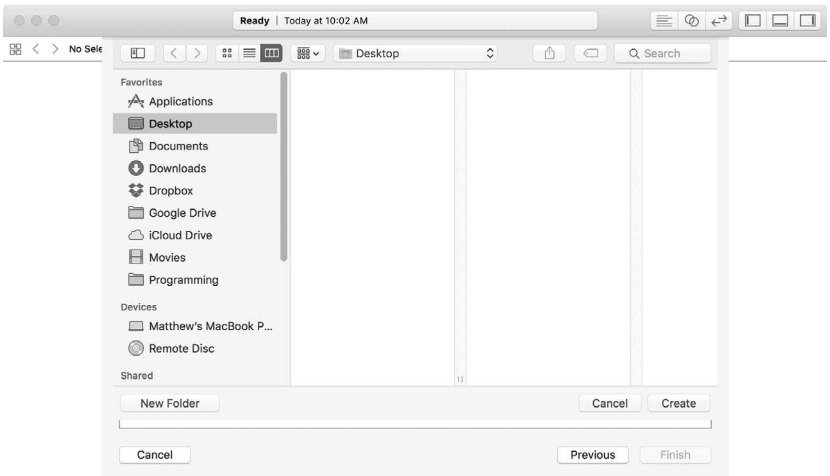


图1-3　保存playground

1.2　尝试 playground

在图1-4中可以看到，Swift的playground打开后有两个区域。左边是Swift代码编辑器，右边是运行结果侧边栏。每次源代码有变化时，就会从上至下运行编辑器中的代码，结果展示在运行结果侧边栏里。

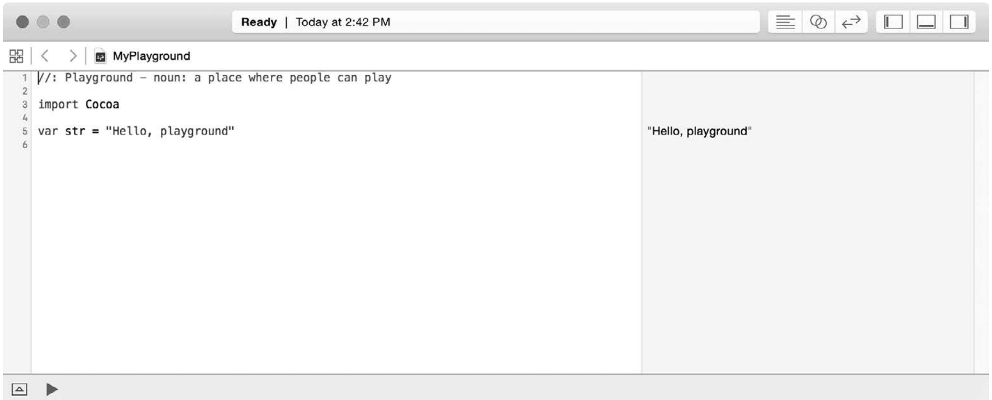


图1-4　新建的playground

来看看新建的playground。注意第一行文本是绿色的，以两个斜杠开头：`//`。斜杠告诉编译器这一行是注释，而Xcode把注释显示为绿色。

开发者可以把注释用作内嵌文档，也可以把它用作笔记来记录这里发生了什么事。斜杠的作用是告诉编译器不要把这一行当作代码。删掉斜杠后编译器会产生一个错误，抱怨自己无法解析表达式。用快捷键Command-/可以很方便地重新加上斜杠（如果是新安装的Xcode，快捷键不起作用的话，重启一下电脑再试试）。

注释下方，playground引入了Cocoa框架。该`import`语句表示playground可以完整访问Cocoa框架中的所有应用程序接口（API）。（API就是说明程序应该如何写的规定或者一组定义。）

`import`语句下面是一行`var str = "Hello, playground"`。引号中的文本被复制到了右边的运行结果侧边栏中：“Hello, playground”。现在仔细看看这行代码。

首先，注意赋值操作符等号。赋值操作符会把它右边的代码结果赋给左边的常量或变量。比如等号左边是文本`var str`。Swift的关键字`var`用来声明变量，下一章会详细介绍这个重要概念。现在，只要知道变量表示你预期会变化的某个值即可。

等号右边是`"Hello, playground"`。Swift中的引号表示字符串（`String`），是字符的有序集合。上面的样板把这个新变量命名为`str`，不过其实几乎可以起任何名字。（当然，也会有限制。试试把`str`改成`var`，看看会发生什么？你觉得为什么不能把变量命名为`var`？请务必把名字改回`str`再往下看。）

现在你能明白右边的运行结果侧边栏中打印的文本是什么了：是赋值给变量`str`的字符串值。

1.3　修改变量并打印信息到控制台

字符串（String）是一种类型，我们将变量str称为“字符串类型的一个实例”。类型是用来表示数据的特殊结构。Swift有很多类型，本书会一一介绍。每种类型都有特定的能力（能对数据做什么）和局限（不能对数据做什么）。比如说，字符串类型旨在处理字符的有序集合，并定义了一系列函数来处理这个字符的有序集合。

回忆一下，str是变量，这意味着你可以改变str的值。我们来给字符串末尾添加一个感叹号，使之成为有标点的完整句子。（在本书中，无论何时添加代码，都会加粗表示；删除则会用删除线表示。）

代码清单1-1　添加标点

```
import Cocoa

var str = "Hello, playground"
str += "!"
```

添加感叹号用到了*加法赋值运算符*+=。加法赋值运算符把加法（+）和赋值（=）运算符组合在了一起。（第3章会详细介绍运算符。）

注意到运算结果侧边栏里的变化了吗？你会看到一行新的结果表示str的新值，它已经补上了感叹号（如图1-5所示）。

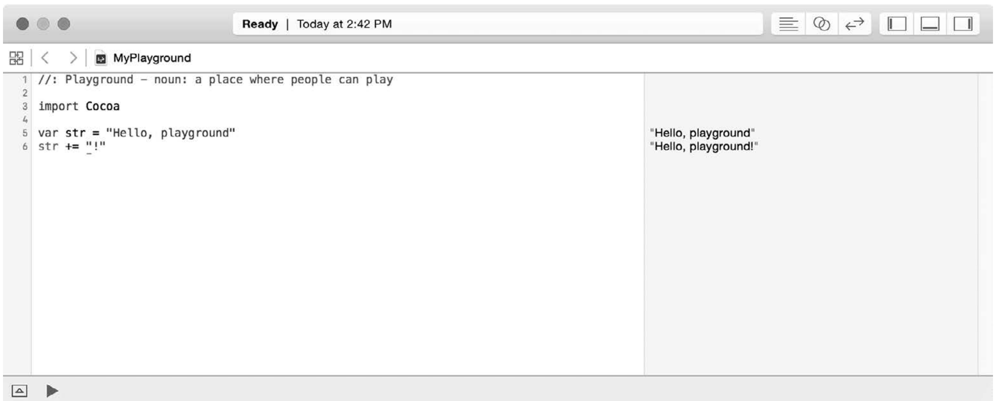


图1-5　修改str

接下来，再加一行代码来把str持有的值打印到*控制台*。在Xcode中，控制台显示你在程序运行过程中创建并输出为日志的文本消息。当发生警告和错误时，Xcode也会用控制台显示。

使用print()函数可以打印信息到控制台。*函数*是一组相关的代码，指示计算机完成特定的任务。print()是打印一个值到控制台然后换行的函数。跟playground不同的是，Xcode项目没有运行结果侧边栏，所以在写功能完备的应用时会频繁用到print()函数。在检查你关注的某个变量的值时，控制台十分有用。

代码清单1-2　打印到控制台

```
import Cocoa

var str = "Hello, playground"
str += "!"
print(str)
```

敲入这行代码并且等playground执行后，Xcode的底部会自动显示控制台（如果没有，打开调试区域就能看到。如图1-6所示，点击View → Debug Area → Show Debug Area。注意到最后一级菜单的快捷键了吗？也可以通过按下Shift-Command-Y来打开调试区域。）

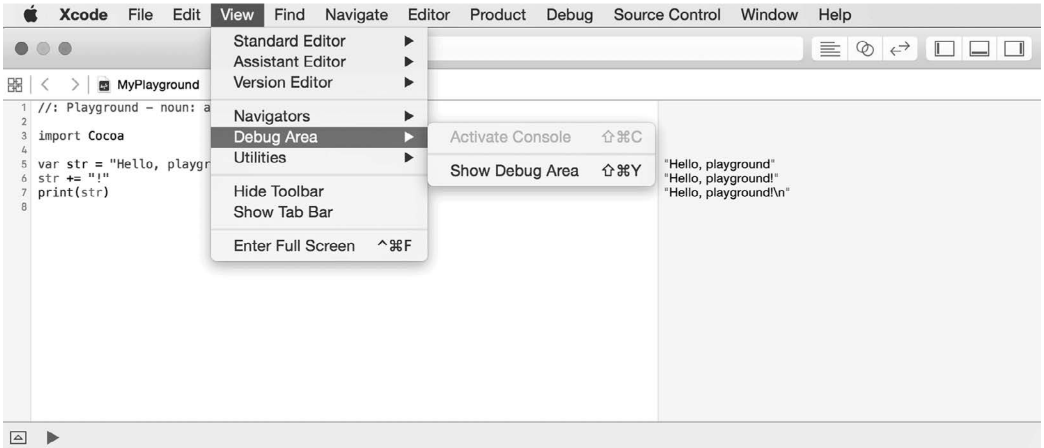


图1-6　显示调试区域

现在调试区域打开了，如图1-7所示。

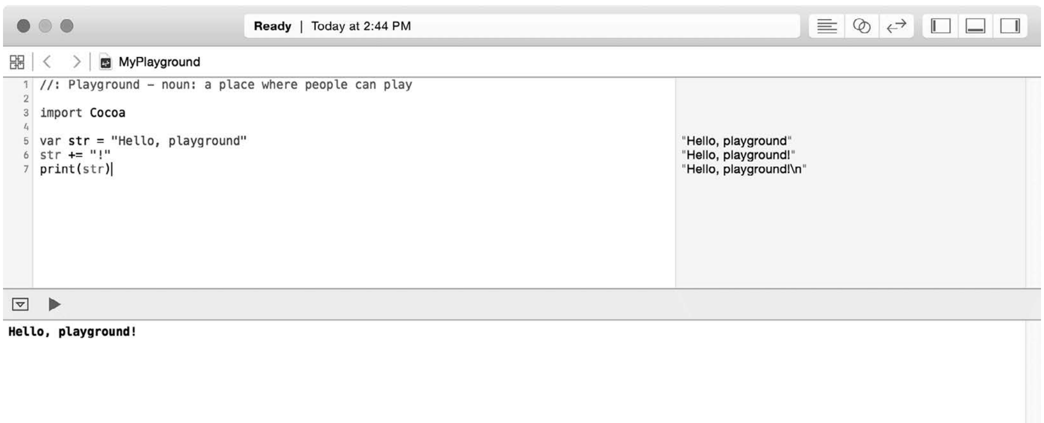


图1-7　第一段Swift代码

1.4 继续前进

现在回顾一下到目前为止的成就，你已经：

- 安装了Xcode；
- 创建并熟悉了playground；
- 使用并修改了变量；
- 知道了字符串类型；
- 用函数打印了一些信息到控制台。

很好！你很快就能开发自己的应用了。坚持下去，随着学习的深入，你会发现本书中几乎所有的内容都不过是基于目前学到的东西稍加变化而来的。

1.5 青铜挑战练习

本书很多章末尾都有一个或多个挑战练习。你要独立完成这些练习以加深对Swift的理解并积攒经验。下面就是你的第一题，不过首先要创建一个新的playground。

你已经知道了字符串类型以及如何用`print()`函数打印信息到控制台。在新的playground里创建一个字符串类型的实例，把实例的值置为你的姓，然后打印到控制台。

第 2 章 类型、常量和变量

本章介绍Swift的基本数据类型、常量和变量。这些元素是构建一切程序的基本构件。常量和变量用来存储值，以及在应用程序中传递数据。类型描述的是常量和变量所持有数据的本质。常量和变量之间、每种数据类型之间都有重要的区别，这些区别形成了它们各自独特的用法。

2.1 类型

变量和常量有数据类型。类型描述数据的本质，并为编译器提供数据处理方式的信息。根据常量或变量的类型，编译器知道该保留多少内存，并且能够进行类型检查（type checking）。这是Swift的一个特性，可以防止你错把其他类型的数据赋值给一个变量。

现在来看看实际操作。创建一个新的macOS playground。（在欢迎画面，选择Get started with a playground；在Xcode内的话，选择File → New → Playground...。）把playground命名为Variables。

假设你想用代码创造一座镇子。你可能想用一个变量表示镇上的红绿灯数量。删除模版自带的代码。创建一个名为`numberOfStoplights`的变量，给它赋个值，如代码清单2-1所示。（记住，要删除的代码用删除线表示。）

代码清单2-1　赋一个字符串给变量

```
import Cocoa

var str = "Hello, playground"
var numberOfStoplights = "Four"
```

这里把字符串类型的实例赋值给了`numberOfStoplights`变量。我们来一步步分析为什么会这样。赋值操作符（`=`）把右边的值赋给左边。Swift用类型推断（type inference）确定变量的数据类型。在上例中，编译器知道变量`numberOfStoplights`的类型是字符串，因为赋值操作符右边的值是字符串的实例。为什么`"Four"`是字符串类型的实例呢？因为引号表示这是字符串字面量。

现在像上一章那样使用+=给变量加上整数2，如代码清单2-2所示。

代码清单2-2　`"Four"`加2

```
import Cocoa
```

```
var numberOfStoplights = "Four"
numberOfStoplights += 2
```

编译器会报错，告诉你这个操作符不能这样用。报错是因为你要把一个数字和字符串相加，而它们的类型不同。把数字2和字符串相加是什么意思？是把字符串重复变成"FourFour"？还是把2添加到末尾变成"Four2"？没人知道。所以把数字和字符串相加是没有意义的。

如果一开始你就认为numberOfStoplights不应该是字符串类型，那么你是对的。因为这个变量表示虚拟镇子上的红绿灯数量，所以用数字类型更合理。Swift提供整型（Int）类型表示整数，显然更适合这个变量。修改代码，改用整型，如代码清单2-3所示。

代码清单2-3　使用数字类型

```
import Cocoa

var numberOfStoplights = "Four"
var numberOfStoplights: Int = 4
numberOfStoplights += 2
```

现在来看看这里的改动。改动之前，编译器靠类型推断来确定变量numberOfStoplights的数据类型。现在则利用Swift的类型注解（type annotation）语法显式地声明变量是整型类型。上面代码中的冒号表示“……是……类型”，所以这行代码可以这么理解：“声明一个名为numberOfStoplights的变量，其类型是Int，初始值是4。”

注意，即使有了类型注解，也不代表编译器不再关注等号两边的类型。举个例子，如果你试图利用类型注解把前面的字符串实例"Four"声明为整数，编译器会警告你字符串无法转化为整数。

Swift有大量常用数据类型。第7章会深入介绍包含文本数据的字符串，第4章则会介绍数字。其他常用类型还有几种容器类型（collection type），本书后面会讲到。

刚才敲入的新代码还有一个变化：错误消失了。在表示镇上红绿灯数量的整数变量上加2完全没有问题。事实上，因为这个实例被声明为变量，所以这样操作非常自然。

2.2　常量与变量

我们说类型描述的是常量或变量所持有数据的本质，那么什么是常量和变量呢？到目前为止，你只见过变量。变量的值可以变化，这意味着你可以给变量赋新值。通过下面这行代码，可以修改numberOfStopligths的值：numberOfStoplights += 2。

不过你经常需要创建值不变的实例。针对这种情况，可以用常量（constant）。顾名思义，常量的值不能改变。

你把numberOfStoplights声明为变量，并且改变了其值。但是如果不想修改numberOfStoplights的值，应该怎么做呢？这种情况下，最好把numberOfStoplights声明为常量。一条很棒的经验法则是：对必须变化的实例用变量，其他都用常量。

在Swift中声明常量和变量有不同的语法。如你所见，声明变量用关键字`var`，而声明常量用关键字`let`。

在当前的playground中声明一个常量来固定镇上的红绿灯数量，如代码清单2-4所示。

代码清单2-4　声明常量

```
import Cocoa

var numberOfStoplights: Int = 4
let numberOfStoplights: Int = 4
numberOfStoplights += 2
```

用`let`关键字声明`numberOfStoplights`是常量。不幸的是，这个改动导致编译器产生了一个错误。为什么呢?

尽管刚才把`numberOfStoplights`声明为了常量，但是还有代码在试图修改其值：`numberOfStoplights += 2`。因为常量无法变化，所以编译器会在你试图修改其值时报错。删除加法赋值那行代码可以修复问题，如代码清单2-5所示。

代码清单2-5　常量无法变化

```
import Cocoa

let numberOfStoplights: Int = 4
numberOfStoplights += 2
```

现在，添加一个整型来表示镇上的人口，如代码清单2-6所示。（你觉得应该用变量还是常量？）

代码清单2-6　声明人口

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
```

镇上的人口会随时间变化，所以用`var`关键字声明人口，使之为变量。还要把人口声明为整型，因为镇上的人口是按整个人计算的。但是这里没有用任何值*初始化*（initialize）人口，所以这是一个未初始化的`Int`。

（初始化是设置一个类型的实例的操作，准备好让其可以使用，我们会在第17章详细介绍。）

用赋值操作符给人口一个初始值，如代码清单2-7所示。

代码清单2-7　给人口赋值

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
```

2.3　字符串插值

每个镇子都要有名字。你的镇子很稳定，所以短期内不会改名字。把镇名声明为字符串常量，如代码清单2-8所示。

代码清单2-8　给镇子命名

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
let townName: String = "Knowhere"
```

如果镇子有简短的描述信息供旅游局使用就好了，下面就来着手解决。描述信息是字符串常量，但是创建方式会跟之前创建的常量和变量有所不同。描述信息要包含刚才输入的所有数据，用Swift的字符串插值（string interpolation）特性来创建。

字符串插值能让你把常量和变量值组合为新字符串。之后，你可以把字符串赋给新变量或常量，也可以打印到控制台。本例会把镇子的描述信息打印到控制台，如代码清单2-9所示。

代码清单2-9　打造镇子的描述信息

```
import Cocoa

let numberOfStoplights: Int = 4
var population: Int
population = 5422
let townName: String = "Knowhere"
let townDescription =
"\(townName) has a population of \(population) and \(numberOfStoplights) stoplights."
print(townDescription)
```

字符串字面量中的`\()`语法表示占位符，可以访问一个实例的值并将其放入新字符串（或者说“插值”）。比如，`\(townName)`访问常量`townName`的值并将其放入新的字符串实例。

新代码的结果如图2-1所示。



图2-1　Knowhere镇的简短描述

2.4　青铜挑战练习

在playground中增加一个表示Knowhere镇失业率水平的变量。你会用什么数据类型？给这个变量设置一个值，并更新`townDescription`来使用这个新信息。

Part 2

第二部分

基础知识

程序代码按特定顺序执行，写软件意味着控制代码执行的顺序。程序语言提供控制流语句（control flow statement）来帮助开发者组织代码执行顺序。为了完成这个任务，第二部分将介绍条件语句和循环的概念。

这部分的几章还会介绍Swift如何在代码中表示数字和文本（通常称为字符串）。这些数据类型是搭建很多应用的构件。学完这几章，你就能对Swift中数字和字符串的用法有一个清晰的理解。

最后，这一部分还会介绍Swift的可空（optional）类型概念。可空类型在程序中扮演重要角色，为程序提供了一种安全表示空（nothing）概念的机制。你将会看到，Swift处理可空类型的方式突显了其写出安全、可靠代码的理念。

第 3 章

条件语句

前几章中的代码就像过家家，只不过声明了几个常量和变量并为其赋值。但是，当一个应用能根据某些变量的值作出决策时，当然就会变得更加贴近生活，而编程就会变得更难一些。比如，一个游戏会让玩家在吃了能量包之后跳过摩天大楼。应用需要用条件语句作出这种决策。

3.1 if/else

if/else语句根据某个特定的逻辑条件执行代码。通常要处理的是一个相对简单的“非此即彼”的状况：根据结果，要么运行一个分支的代码，要么运行另一个分支的代码（但是不会同时运行两条分支的代码）。

想象你在上一章中的Knowhere镇里，需要买邮票。Knowhere要么有邮局，要么没有。如果有邮局，就在邮局买邮票；如果没有，那就得开车去隔壁镇上买。有没有邮局就是逻辑条件，而不同的行为就是“在镇上买邮票”和“去镇外买邮票”。

有些条件比二元的是非更复杂，第5章就会介绍一种叫作switch的复杂机制。不过现在先看简单的情形。

新创建一个新的macOS playground，命名为Conditionals。敲入代码清单3-1所示的代码，这是if/else语句的基本语法。

代码清单3-1　大还是小

```
import Cocoa

var str = "Hello, playground"
var population: Int = 5422
var message: String

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
print(message)
```

这段代码首先声明population为整型并为其赋值5422，还声明了String类型的变量

message。一开始不初始化这个变量，即不赋值。

接下来是条件语句if/else,这里会根据if语句的计算结果是否为真来给message赋值。（注意这里用了字符串插值来把人口信息放进message字符串。）

图3-1是现在playground大概的样子。控制台和运行结果侧边栏显示message的值是当条件语句为真时的字符串字面量。这是怎么做到的？

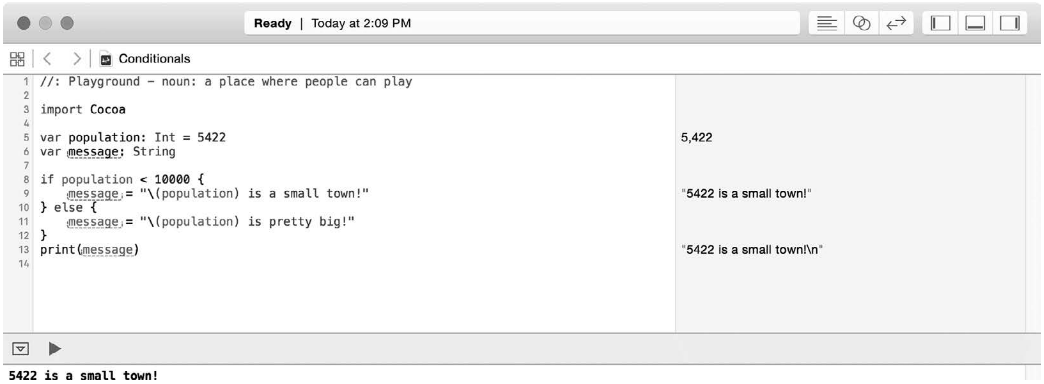


图3-1 根据不同情况描述镇子人口

if/else语句中的条件用比较运算符<测试镇子人口是否小于10 000。如果条件计算为真，message被置为第一个字符串字面量（“X is a small town!”）。如果条件计算为假（人口大于或等于10 000），则message被置为第二个字符串字面量（“X is pretty big!”）。在本例中，镇子人口小于10 000，所以message被置为“5422 is a small town!”。

表3-1列出了Swift的比较运算符。

表3-1 比较运算符

<	计算左边的值是否小于右边的值
<=	计算左边的值是否小于等于右边的值
>	计算左边的值是否大于右边的值
>=	计算左边的值是否大于等于右边的值
==	计算左边的值是否等于右边的值
!=	计算左边的值是否不等于右边的值
===	计算两个实例是否指向同一个引用
!==	计算两个实例是否不指向同一个引用

现在还不需要理解所有运算符的描述。在阅读本书的过程中，你会遇到很多运算符的实际用法，而且会随着使用经验的增加对它们有更清楚的认识。如果有问题就回来查查这张表。

有时候我们只关心条件计算的一个结果，也就是说，当某个条件满足时执行一段代码，否则就什么都不做。敲入代码清单3-2所示的代码。（注意两个地方有新增的代码，都加粗了。）

代码清单3-2　镇上有邮局吗

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

这里新增了一个叫作`hasPostOffice`的变量，其类型是布尔（Boolean，`Bool`）。布尔类型的值是二选一的：真或假。在本例中，布尔变量`hasPostOffice`记录镇上是否有邮局。这里置为真，也就是有邮局。

`!`是逻辑运算符（logical operator），称为“逻辑非”。它能测试`hasPostOffice`是否为假。你可以把`!`理解为反转一个布尔值：真变为假，假变为真。

上面的代码先把`hasPostOffice`置为真，然后看其是否为假。如果`hasPostOffice`为假，你就不知道去哪里买邮票，所以提出了问题。如果`hasPostOffice`为真，你就知道去哪里买邮票，不需要问别人，所以什么都不会发生。

镇子确实有邮局（因为`hasPostOffice`初始化为真），所以条件`!hasPostOffice`为假。也就是说，`hasPostOffice`为假的情况没有发生，所以`print()`函数永远不会被调用。

表3-2列出了Swift的逻辑运算符。

表3-2　逻辑运算符

`&&`	逻辑与：当且仅当两者都为真时结果为真（否则结果为假）
`||`	逻辑或：两者任意之一为真时结果为真（只有两者都为假时结果为假）
`!`	逻辑非：真变为假，假变为真

3.2　三目运算符

三目运算符（ternary operator）跟`if/else`很像，但是语法更简洁：`a ? b : c`。在汉语中，三目运算符可以这么念：“如果a为真，那么做b；否则做c。”

现在用三目运算符改写用`if/else`检查镇上人口的代码，如代码清单3-3所示。

代码清单3-3　使用三目运算符

```
...
```

```
~~if population < 10000 {~~
    ~~message = "\(population) is a small town!"~~
~~} else {~~
    ~~message = "\(population) is pretty big!"~~
~~}~~

message = population < 10000 ? "\(population) is a small town!" :
                     "\(population) is pretty big!"
...
```

三目运算符可能会成为争议的源头：有些程序员喜欢，有些程序员则厌恶。我们抱一种中立态度。这里的用法算不上优雅，为message赋值需要一个比a ? b : c更复杂的语句。三目运算符很适合简洁的语句，但是如果代码开始折行，我们认为应该改用if/else。

用快捷键Command-Z来撤销刚才的改动，把三目运算符去掉，恢复if/else语句，如代码清单3-4所示。

代码清单3-4　恢复if/else

```
...
~~message = population < 10000 ? "\(population) is a small town!" :~~
                     ~~"\(population) is pretty big!"~~
if population < 10000 {
    message = "\(population) is a small town!"
} else {
    message = "\(population) is pretty big!"
}
...
```

3.3　嵌套的 if

对于大于两种可能性的场景，可以嵌套if语句，通过把一个if/else语句写在另一个if/else语句的花括号里实现。来看个例子，我们在现有的if/else语句的else块中嵌套一个if/else语句，如代码清单3-5所示。

代码清单3-5　嵌套条件语句

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
        message = "\(population) is pretty big!"
```

```
    }
}
print(message)

if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

嵌套的`if`分支用到了比较运算符`>=`和逻辑运算符`&&`来检查人口是否在10 000~50 000的区间内。因为你的镇子人口没有落在这个区间，所以跟之前一样，`message`被置为“5422 is a small town!”。

尝试增加人口来试试别的分支。

在编程中嵌套`if/else`很常见，你会在实际工作中遇到，自己也会这么写。对于嵌套多深是没有限制的，不过嵌套过深的危险在于会让代码难以读懂。一两层没事，超过两层代码就不易读也不好维护了。

有一些方法可以避免嵌套语句。下面我们来重构（refactor）刚才的代码，使之变得易懂。重构的意思是修改代码，使之以不同的方式做同样的事情。重构后代码可能会变得更高效、更易懂或者只是变得更好看。

3.4　else if

`else if`条件语句可以让你把多个条件语句串起来。它能让你检查多种情况，并根据哪个分支为真来执行代码。你可以把任意多个`else if`串起来，不过只有一种条件会得到匹配。

为了让代码更易读，把嵌套的`if/else`提取出来变成一个独立的分支，用来判断镇子是否为中等规模，如代码清单3-6所示。

代码清单3-6　使用else if

```
import Cocoa

var population: Int = 5422
var message: String
var hasPostOffice: Bool = true

if population < 10000 {
    message = "\(population) is a small town!"
} else if population >= 10000 && population < 50000 {
    message = "\(population) is a medium town!"
}  else {
    if population >= 10000 && population < 50000 {
        message = "\(population) is a medium town!"
    } else {
    message = "\(population) is pretty big!"
   }
}
print(message)
```

```
if !hasPostOffice {
    print("Where do we buy stamps?")
}
```

这里只用了一个`else if`分支，不过有需要的话可以串很多个。这段代码比上面的嵌套`if/else`有所改善。如果你发现自己用了大量的`if/else`语句，那么也许最好用另外一种机制来实现，比如第5章将会讲到的`switch`，敬请期待。

3.5 青铜挑战练习

给判断镇子规模的代码再增加一个`else if`语句，用来看镇上人口规模是否非常大。自己选择一个人口阈值，相应地设置`message`变量。

第4章 数

数是计算机语言的基础，也是软件开发的重要组成部分。数用来记录温度高低，判断一个句子中有多少字母，以及镇上有多少僵尸出没。数分为两种基本类型：整数和浮点数。

4.1 整数

前面我们用过整数，但是还没有对其下一个定义。整数是没有小数点和小数部分的数，也就是整个的数。整数常用来表示事物的数量，比如一本书的页数。计算机所用的整数和你在别处用的整数有一个区别，那就是计算机的整数类型占据固定大小的内存，所以无法表示所有的整数——它们有最小值和最大值。

我们可以给出最小值和最大值，不过还是让Swift来告诉你答案吧。创建一个新的playground，命名为Numbers，敲入代码清单4-1所示的代码。

代码清单4-1　Int的最大值和最小值

```
import Cocoa

var str = "Hello, playground"

print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
```

你会在控制台看到如下输出：

```
The maximum Int value is 9223372036854775807.
The minimum Int value is -9223372036854775808.
```

为什么这两个数分别是最小和最大的Int值呢？计算机用固定位数二进制的形式保存整数，1位就是一个0或1，每个位的位置表示2的不同次幂。要计算一个二进制数的值，只要找到每个存储了1的位，将其对应的2的次幂相加即可。举个例子，38和–94的8位有符号整数二进制表示如图4-1所示。［注意，位的顺序是从右往左数的。有符号（signed）意味着这个整数既可以表示正数又可以表示负数。稍后会详细介绍有符号整数。］

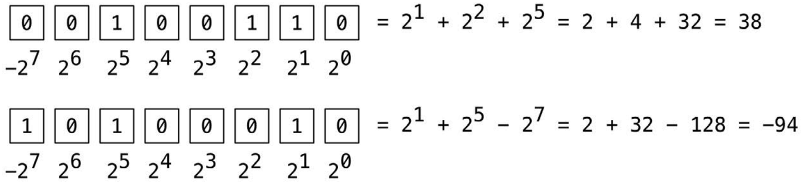


图4-1　二进制数

在macOS中，`Int`是64位整数，也就是说它能表示2^{64}种可能的值。想象一下图4-1中的数是64位而不是8位，那么最高位（最左边）表示的2的次幂将是-2^{63}=–9 223 372 036 854 775 808，正好是你在playground中看到的`Int.min`的值。如果你把2^0、2^1、...、2^{62}相加，最终会得到9 223 372 036 854 775 807，正好是`Int.max`的值。

在iOS中，`Int`更复杂一些。苹果从iPhone 5S、iPad Air和带Retina屏的iPad mini开始引入64位设备，而更早的设备还是32位架构。如果你在给新设备写应用，也就是"以64位架构为目标"，`Int`就是64位的，跟macOS一样。另一方面，如果你是以32位架构为目标，比如iPhone 5或者iPad 2，那么`Int`就是32位整数。编译器会在构建程序时选择合适的`Int`长度。

如果需要知道整数的精确长度，可以使用Swift的显式长度整数类型，比如`Int32`是Swift的32位有符号整型。用`Int32`来看看32位整数的最小和最大值，如代码清单4-2所示。

代码清单4-2　`Int32`的最大和最小值

```
...
print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
print("The maximum value for a 32-bit integer is \(Int32.max).")
print("The minimum value for a 32-bit integer is \(Int32.min).")
```

对于8位、16位和64位有符号整型，相应的还有`Int8`、`Int16`、`Int64`。需要知道整数长度时就用这些带长度的整型，比如使用某些算法（加密算法中比较常见）或者需要和其他计算机交换整数时（比如通过互联网发送数据）。这些类型不太常用，良好的Swift编程风格是在大部分情况下用`Int`。

到目前为止，我们所见的整数类型都是有符号的，也就是说它们既能表示正数又能表示负数。Swift也有无符号整型，用来表示大于等于0的整数。每个有符号整型（`Int`、`Int16`等）都有对应的无符号整型（`UInt`、`UInt16`等）。有符号整数和无符号整数的区别在于二进制层面最高位（对于8位整数来说是2^7）表示的2的次幂是正数还是负数。我们来看代码清单4-3所示的几个例子。

代码清单4-3　无符号整数的最大和最小值

```
...
print("The maximum Int value is \(Int.max).")
print("The minimum Int value is \(Int.min).")
print("The maximum value for a 32-bit integer is \(Int32.max).")
print("The minimum value for a 32-bit integer is \(Int32.min).")
```

```
print("The maximum UInt value is \(UInt.max).")
print("The minimum UInt value is \(UInt.min).")
print("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
print("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")
```

就像Int一样，UInt在macOS中是64位整数，而在iOS中则取决于目标硬件，可能是32位也可能是64位。所有无符号整型的最小值都是0，而N位的无符号整型最大值是2^N-1。比如说，64位无符号整型的最大值是$2^{64}-1$，等于18 446 744 073 709 551 615。

有符号整型和无符号整型的最小值和最大值之间有对应关系：UInt64的最大值是Int64的最大值与最小值绝对值之和。无论是有符号整型还是无符号整型，都有2^{64}种可能的值，但是有符号整型需要把一半可能的值留给负数用。

有些数看起来很自然地应该用无符号整数表示，比如物体的个数不可能是负数。不过Swift的风格更倾向于在所有的整数场景中（包括数量）使用Int，除非算法或代码明确要求用无符号整数。要理解为什么推荐始终用Int涉及本章后面要讨论的话题，所以稍后再解释原因。

4.2　创建整数实例

第2章中已经创建过Int的实例，还讲到了可以显式或隐式地声明类型，如代码清单4-4所示。

代码清单4-4　显式和隐式地声明Int

```
...
print("The maximum value for a 32-bit unsigned integer is \(UInt32.max).")
print("The minimum value for a 32-bit unsigned integer is \(UInt32.min).")

let numberOfPages: Int = 10 // 显式声明类型
let numberOfChapters = 3    // 还是Int类型，不过是编译器推断出来的
```

编译器总是假设隐式声明的整数值是Int，不过你也可以用显式声明创建其他整数类型的实例，如代码清单4-5所示。

代码清单4-5　显式声明其他整数类型

```
...
let numberOfPages: Int = 10 // 显式声明类型
let numberOfChapters = 3    // 还是Int类型，不过是编译器推断出来的

let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = -1000
```

如果用一个非法的值创建实例会怎么样？比如说，如果用负数值创建UInt或者用大于127的值创建Int8会怎么样？参照代码清单4-6来试试看。

代码清单4-6　用非法的值创建整数类型

```
...
let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = -1000
```

```
// 接下去要有麻烦了！
let firstBadValue: UInt = -1
let secondBadValue: Int8 = 200
```

你应该会在playground的左侧看到红色感叹号。点击感叹号可以看到错误信息（如图4-2所示）。

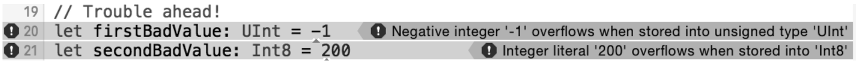


图4-2　整数溢出错误

编译器报告，你输入的两个值在保存到Uint和Int8类型的常量中时溢出了。“在保存到……中时溢出了”的意思是，当编译器尝试把数存入你指定的类型时，超出了该类型所允许的值的范围。Int8可以保存从–128到127的值，而200超出了这个范围，所以试图把200存入Int8会导致溢出。

删除有问题的代码，如代码清单4-7所示。

代码清单4-7　删除错误的值

```
...
// 接下去要有麻烦了！
let firstBadValue: UInt = -1
let secondBadValue: Int8 = 200
```

4.3　整数操作符

Swift允许利用人们熟悉的加（+）、减（-）、乘（*）、除（/）操作符对整数做基本算数运算。尝试打印一些算式结果，如代码清单4-8所示。

代码清单4-8　基本运算操作符

```
...
let numberOfPeople: UInt = 40
let volumeAdjustment: Int32 = -1000

print(10 + 20)
print(30 - 5)
print(5 * 6)
```

编译器遵从数学运算的优先级（precedence）和结合性（associativity）法则，二者定义了单个表达式中多个操作符的运算顺序。代码清单4-9是一个例子。

代码清单4-9　操作符运算顺序

```
...
print(10 + 20)
print(30 - 5)
```

```
print(5 * 6)

print(10 + 2 * 5) // 20，因为 2 * 5先计算
print(30 - 5 - 5) // 20，因为30 - 5先计算
```

你可以选择记住优先级和结合性法则，不过我们推荐更简单的方法，用圆括号来让你的意图变得明确，因为圆括号总是最先得到计算，如代码清单4-10所示。

代码清单4-10　圆括号是你的朋友

```
...
print(10 + 2 * 5) // 20，因为 2 * 5先计算
print(30 - 5 - 5) // 20，因为30 - 5先计算
print((10 + 2) * 5) // 60，因为(10 + 2)先计算
print(30 - (5 - 5)) // 30，因为(5 - 5)先计算
```

4.3.1　整数除法

表达式`11 / 3`的值是多少？你可能（有理由）认为是3.666 666 666 67，不过还是参照代码清单4-11试试看。

代码清单4-11　整数除法的结果可能出人意料

```
...
print((10 + 2) * 5) // 60，因为(10 + 2)先计算
print(30 - (5 - 5)) // 30，因为(5 - 5)先计算

print(11 / 3) // 打印3
```

两个整数的操作结果永远是整数，3.666 666 666 67不是整数，所以Swift把小数部分截断后留下了3。如果结果是负数，比如`-11 / 3`，小数部分还是会被截断，结果是–3。因此，整数除法总是向0舍入。

有时候取余数比较有用，取余操作符（`%`）就是返回余数用的，如代码清单4-12所示。（如果你熟悉数学中的模运算以及其他的编程语言，那么要小心了：取余操作符不太一样，用在负整数上的结果可能会出乎你的意料。）

代码清单4-12　余数

```
...
print(11 / 3) // 打印3
print(11 % 3) // 打印2
print(-11 % 3) // 打印-2
```

4.3.2　快捷操作符

目前为止所见的操作符都会返回一个新值。不过这些操作符都有原地修改变量的版本。编程中很常见的一种操作是给一个整数加上或减去一个整数，也可以用+=或`-=`，前者是加法和赋值操

作的组合，后者是减法和赋值操作的组合。

代码清单4-13　把加法或减法和赋值组合

```
...
print(11 % 3) // 打印2
print(-11 % 3) // 打印-2

var x = 10
x += 10 // 等同于: x = x + 10
print("x has had 10 added to it and is now \(x)")
x -= 5 // 等同于: x = x - 5
print("x has had 5 subtracted from it and is now \(x)")
```

对于其他的基本算术运算，也有相应的运算和赋值组合的快捷操作符：*=、/=和%=。每个都会把运算结果赋给操作符左边的值。

4.3.3　溢出操作符

你认为代码清单4-14中z的值是多少？（在敲入代码并得到明确答案之前先想想清楚。）

代码清单4-14　z的值

```
...
let y: Int8 = 120
let z = y + 10
```

如果你认为z的值是130，那你不是唯一一个这么想的人。但是输入代码后你会发现Xcode报错了。点击感叹号查看详细信息（如图4-3所示）。

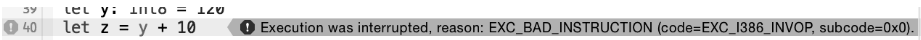


图4-3　在Int8上做加法时执行中断了

“执行中断”是什么意思？我们来分解每一步发生了什么：

(1) y是Int8类型，所以编译器认为y + 10也必须是Int8；

(2) 编译器因而推断z是Int8；

(3) playground运行后，Swift给y加10，得到130；

(4) 在把结果存入z之前，Swift检查发现130对于Int8来说是非法值。

但是Int8只能保存从–128到127的值，130太大了！因此playground触发了*陷阱*，程序停止运行。我们会在第20章详细介绍陷阱。目前，只要知道陷阱会导致程序立即停止运行并输出错误消息就行了，这表示发生了严重问题需要你检查。

Swift提供*溢出操作符*（overflow operator），它们在值太大（或太小）时的行为不同于普通操作符，会绕过去而不是触发陷阱。实际运行一下代码来看看这是什么意思。溢出加操作符是&+，参照代码清单4-15，用它来替换代码中的加法。

代码清单4-15　使用溢出操作符

```
...
let y: Int8 = 120
let z = y + 10
let z = y &+ 10
print("120 &+ 10 is \(z)")
```

对120 + 10进行溢出加操作并存入Int8的结果是–126。这是你预期的结果吗？

可能不是。（不过也没事！）要理解这个结果的逻辑，想象每次给y加1。因为y是Int8类型的，所以一旦达到127就不能再往上加了。在加1之后不会得到128，而是折回到了–128。因此120 + 8 = –128，120 + 9 = –127，120 + 10 = –126。

减法和乘法也有溢出版本的操作符：&-和&*。乘法操作符有溢出版本很好理解，但是减法呢？减法很明显不会上溢，但是可能会下溢（underflow）。比如说，一个值为–120的Int8减去10会让结果太小，无法用Int8表示。用&-会让下溢折回到正数，得到126。

整数操作的上溢和下溢可能是很多难以追查的严重bug的根源。Swift的设计理念是安全第一，尽量减少错误。如果你有其他编程语言经验，Swift对于计算溢出默认触发陷阱的行为可能会让你大吃一惊，因为其他大多数语言的默认行为都是折回，跟Swift的溢出操作符一样。Swift的哲学是触发陷阱（即使会导致程序崩溃）比潜在的安全漏洞要好。不过算数运算溢出折回在一些情况下也是有用的，所以有需要的话可以使用这些特殊操作符。

4.4　转换整数类型

目前所见的操作符都是在两个类型完全一样的值之间进行操作。如果像代码清单4-16那样，试图操作两个不同类型的数会怎么样？

代码清单4-16　两个不同类型的值相加

```
...
let a: Int16 = 200
let b: Int8 = 50
let c = a + b // 啊哦！
```

这里会出现编译错误。a和b不能相加，因为它们类型不同。有些语言会在做这类操作时自动进行类型转换，但是Swift不会，你得自己手动转换类型使之匹配。

在这个例子中，要么把a转换成Int8，要么把b转换成Int16。不过事实上，只有一种会成功，如代码清单4-17所示。（为什么？再读一下上一节！）

代码清单4-17　转换类型使加法可以操作

```
...
let a: Int16 = 200
let b: Int8 = 50
let c = a + b // 啊哦！
let c = a + Int16(b)
```

现在我们可以回到一开始的建议了：在Swift中大部分需要使用整数的场景都应该坚持使用Int，即使是那些只有正数才有意义的值（比如事物的个数）。Swift对于字面量的默认推断类型是Int，而不同整数类型之间如果不做转换的话默认是不能操作的。在写代码过程中一直使用Int可以极大减少类型转换的需要，也能让你自如地对整数使用类型推断。

Swift区别于其他编程语言的另一个特性是它需要你（程序员）来决定在不同类型间做数学运算时应该怎么进行类型转换。这个要求还是为了保证编程的安全和正确性。比如，为了对不同类型的数做数学运算，C编程语言会转换不同类型，但是这种转换有时候会丢失精度，也就是说在转换过程中丢掉某些信息。在不同类型的数之间做数学运算的Swift代码会更冗长，但是很容易看明白做了什么转换。增加一点代码能让你更容易明白并维护代码。

4.5 浮点数

为了表示有小数点的数（比如3.2），需要用到*浮点数*（floating-point number）。关于浮点数，有两件事要牢记在心。首先，浮点数在计算机中是以*尾数*（mantissa）和*指数*（exponent）的形式存储的，类似于科学记数法。比如说，123.45可以用1.2345×10^2的形式或12.345×10^1的形式存储（尽管计算机使用二进制而不是十进制）。其次，浮点数通常不精确：有很多数无法以浮点数的形式精确存储。计算机会存储一个近似值，非常接近你要的数。（稍后会详细介绍。）

Swift有两种基本的浮点数类型：32位浮点数Float和64位浮点数Double。Float和Double的长度差异并不像整数那样影响其最小值和最大值，而是影响其精度。Double的精度比Float高，这意味着它能存储更精确的近似值。

在Swift中，浮点数的默认推断类型是Double，就像整数的不同类型一样，你也可以显式地声明Float和Double，如代码清单4-18所示。

代码清单4-18 声明浮点数类型

```
...
let d1 = 1.1 // 隐式Double声明
let d2: Double = 1.1
let f1: Float = 100.3
```

所有数字操作符对浮点数都适用（除了取余操作符只能用于整数），如代码清单4-19所示。

代码清单4-19 浮点数操作符

```
...
let d1 = 1.1 // 隐式Double声明
let d2: Double = 1.1
let f1: Float = 100.3

print(10.0 + 11.4)
print(11.0 / 3.0)
```

你应该时刻记得浮点数和整数的重大差别就是浮点数天生是不精确的。举个例子，还记得第3章的==操作符吧，它会判断两边的值是否相等。你可能会预期浮点数也能用==来比较，如代码

清单4-20所示。

代码清单4-20　比较两个浮点数

```
...
print(10.0 + 11.4)
print(11.0 / 3.0)

if d1 == d2 {
    print("d1 and d2 are the same!")
}
```

d1和d2都初始化为1.1了，目前为止一切正常。现在给d1加0.1，你会预期其结果等于1.2，所以把结果跟1.2比较，如代码清单4-21所示。

代码清单4-21　出人意料的结果

```
if d1 == d2 {
    print("d1 and d2 are the same!")
}

print("d1 + 0.1 is \(d1 + 0.1)")
if d1 + 0.1 == 1.2 {
    print("d1 + 0.1 is equal to 1.2")
}
```

这个结果可能会让你大吃一惊！从第一个print()函数的输出应该能看到d1 + 0.1的结果是1.2，但是在if语句中的print()函数却没有运行。为什么？难道1.2不等于1.2？好吧，有时候等于，有时候不等于。

正如我们之前说的，很多数（包括1.2）无法用浮点数精确表示。计算机会存储一个非常接近1.2的近似值，当你给1.1加0.1后，结果实际上是类似于1.200 000 000 000 000 1的值，而你输入字面量1.2后存储的值类似于1.199 999 999 999 999 9。尽管在打印的时候Swift会把两者都舍入为1.2，但是从技术上说它们并不相等，所以在if语句中的print()函数没有执行。

浮点数背后令人头疼的细节超出了本书的范围。这个例子告诉我们浮点数有些潜在陷阱。因此，永远不要用浮点数表示那些必须精确的数值（比如金额计算），有一些别的工具可以做这类事。

4.6　青铜挑战练习

放下计算机，拿起纸笔来做这个练习。–1的8位有符号整数二进制表示是多少？

如果还是这个二进制串，但是把它解释为一个8位无符号整数，其值是多少？

第 5 章

switch语句

在第3章，我们见识了一种条件语句：`if/else`。在学习过程中曾提到，`if/else`对于不止一个条件的场景来说不太够用。本章就要来看看`switch`语句了。与`if/else`不同，`switch`非常适于处理多重条件。你会看到Swift的`switch`语句是这门语言非常灵活、强大的特性。

5.1　什么是 switch

`if/else`语句会根据我们关注的条件是否为真来执行代码。与之对应的是，`switch`语句关注某个特殊的值，并将其与一系列分支（case）进行匹配；如果能匹配上，`switch`就会执行对应的代码。下面是`switch`的基本语法。

```
switch aValue {
case someValueToCompare:
    // 做一些响应操作

case anotherValueToCompare:
    // 做一些响应操作

default:
    // 没有匹配时的操作
}
```

在上例中，`switch`只和两个分支做了比较，但是它能包含任意数量的分支。如果`aValue`匹配了任意一个参与比较的分支，那个分支的代码就会被执行。

注意`default`分支的使用，当参与比较的值没有匹配到任何分支时就会执行这个分支。`default`分支不是必需的，不过对于`switch`语句，被检查类型的每个值都必须有相应的分支。所以用`default`分支通常更高效，这样就不需要为该类型的每个值都提供一个特定的分支了。

你可能已经猜到了，为了能进行比较，每个分支的类型都必须和被比较的类型一样。换句话说，`aValue`的类型必须与`someValueToCompare`和`anotherValueToCompare`的类型一样。

这段代码展示了`switch`语句的基本语法，但是不符合语法规则。事实上，这个`switch`语句会引发编译错误。为什么呢？如果你想知道，可以把代码敲入playground看一看。给`aValue`和所有的分支赋上值，你会看到每个分支都有错误："'case' label in a 'switch' should have at least one executable statement.（`switch`语句中的`case`标签下至少要有一个可执行的语句。）"

问题出在每个分支都至少要有一行可执行的代码，这是switch语句的目的：每个分支代表一个单独的代码执行分支。在本例中，分支下面只有注释，而注释无法执行，所以这个switch语句没有满足要求。

5.2　开始使用 switch

创建一个名为Switch的playground，并搭建好switch代码框架，如代码清单5-1所示。

代码清单5-1　初次使用switch

```
import Cocoa

var str = "Hello, playground"

var statusCode: Int = 404
var errorString: String
switch statusCode {
case 400:
    errorString = "Bad request"

case 401:
    errorString = "Unauthorized"

case 403:
    errorString = "Forbidden"

case 404:
    errorString = "Not found"

default:
    errorString = "None"
}
```

为了匹配到能描述错误信息的String实例，上面的switch语句把一个HTTP状态码跟四个分支做了比较。因为404分支与statusCode匹配，所以errorString被赋值为"Not found"，在侧边栏中可以看到（如图5-1所示）。试着改变statusCode的值来查看其他结果，看完再设置回404。

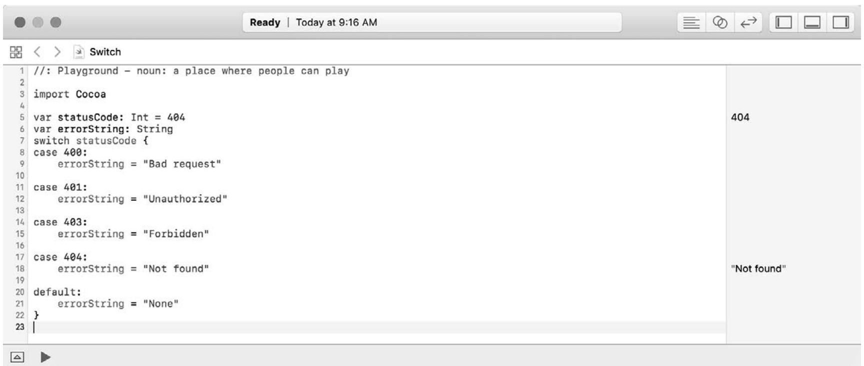


图5-1　匹配错误字符串和状态码

假设你想用switch语句生成一个有意义的错误描述，可以参照代码清单5-2修改代码。

代码清单5-2　switch分支可以有多个值

```
import Cocoa

var statusCode: Int = 404
var errorString: String = "The request failed:"
switch statusCode {
case 400:
    errorString = "Bad request"

case 401:
    errorString = "Unauthorized"

case 403:
    errorString = "Forbidden"

case 404:
    errorString = "Not found"

default:
    errorString = "None"
case 400, 401, 403, 404:
    errorString = "There was something wrong with the request."
    fallthrough
default:
    errorString += " Please review the request and try again."
}
```


现在，一个分支覆盖了所有的错误状态码（用逗号隔开了）。如果statusCode匹配了这个分支中的任何一个值，文本"There was something wrong with the request."就会被赋给errorString。

上面的代码中还添加了名为fallthrough的*状态转移语句*（control transfer statement）。状态转移语句能让你修改某个控制流中的代码执行顺序。这类语句能把控制权从一块代码转移到另一块代码。第6章中关于循环的内容还会讲到另一种控制转移语句用法。

在这里，fallthrough告诉switch语句从一个分支的底部“漏”（fall through）到下一个分支去。如果某个匹配上的分支末尾有fallthrough控制转移语句，它会先执行自己的代码，再把控制权转移到下面紧挨着的分支。紧挨着的分支又会执行自己的代码，无论它跟正在检验的值是否匹配；如果那个分支末尾也有fallthrough语句，它就会把控制权传递给再下一个分支，以此类推。fallthrough语句能让你不需要进行匹配就能进入一个分支并执行其代码。

本例中，由于fallthrough语句的存在，虽然第一个分支匹配上了，但是switch语句不会停止执行，它会继续处理default分支。如果没有fallthrough关键字，switch语句就会在第一次匹配成功后停止执行。本例使用fallthrough可以帮助构建errorString，同时避免使用奇怪的逻辑来确保比较的值能匹配上我们关注的这些分支。

default分支使用了复合赋值操作符（+=）来给errorString添加检查请求的建议。这个

switch语句最终的结果是把errorString设置为"There was something wrong with the request. Please review the request and try again."。如果提供的状态码不能匹配到分支中的值，最终的结果是errorString被设置为"Please review the request and try again."。

如果熟悉诸如C或者Objective-C之类的其他语言，你会发现Swift中switch语句的工作方式有所不同。那些语言中的switch语句会自动穿透分支，需要在分支代码的末尾添加一个break控制转移语句来跳出switch。Swift的switch语句则正好相反：如果匹配上一个分支，那么这个分支执行代码，然后switch停止运行。

5.2.1 区间

前面介绍了每个分支只有一个值与给定值比较的switch语句，以及每个分支有多个值与给定值比较的switch语句。switch语句也可以用valueX...valueY这样的语法来把某个区间内的值与给定值比较。参照代码清单5-3修改代码，来看看其实际用法。

代码清单5-3 switch分支可以用一个值、多个值或者区间值

```
import Cocoa

var statusCode: Int = 404
var errorString: String = "The request failed with the error:"
switch statusCode {
    case 400, 401, 403, 404:
        errorString += " There was something wrong with the request."
        fallthrough
    default:
        errorString += " Please review the request and try again."
}

switch statusCode {
case 100, 101:
    errorString += " Informational, 1xx."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, 3xx."

case 400...417:
    errorString += " Client error, 4xx."

case 500...505:
    errorString += " Server error, 5xx."

default:
    errorString = "Unknown. Please review the request and try again."
}
```

上面的`switch`语句利用了区间匹配（range matching）语法`...`为HTTP状态码创建了闭区间，也就是说，`300...307`是包含了300、307以及其间所有整数的区间。

上面代码中还有单个HTTP状态码的分支（第二个），显式列出的用逗号分隔两个状态码的分支（第一个），以及一个默认分支。它们的形式和前面介绍过的分支一样。所有的分支语法都可以在一个`switch`语句中组合使用。

这个`switch`语句的结果是`errorString`被设置为`"The request failed with the error: Client error, 4xx."`。再试试改变`statusCode`的值来查看别的结果，但是继续往下读之前务必把它改回`404`。

5.2.2 值绑定

假设无论程序能否识别状态码，都需要把实际的状态码放进`errorString`，那么可以利用Swift的值绑定（value binding）特性在前面`switch`语句的基础上做到这一点。

值绑定能在某个特定分支中把待匹配的值绑定（bind）到本地的常量或变量上。这个常量或变量只能在该分支中使用，如代码清单5-4所示。

代码清单5-4 使用值绑定

```
...
switch statusCode {
case 100, 101:
    errorString += " Informational, 1xx."
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, 3xx."
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, 4xx."
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, 5xx."
    errorString += " Server error, \(statusCode)."

default:
    errorString = "Unknown. Please review the request and try again."

case let unknownCode:
    errorString = "\(unknownCode) is not a known error code."
}
```

这里用了字符串插值来把`statusCode`传入每个分支的`errorString`。

仔细看一下最后一个分支。当statusCode没有匹配上面的任何一个分支时，我们创建了一个临时常量unknownCode，将其绑定为statusCode的值。举个例子，如果statusCode的值等于200，那么switch会把errorString设为"200 is not a known error code."。因为unknownCode会把任何没有匹配上前面分支的值拿过来，所以也就不需要显式的默认分支了。

注意这里用了常量，所以unknownCode的值是固定的。如果因为某些原因需要对unknownCode做些处理，就可以用var而不是let来声明。举例来讲，这么做可以在最后一个分支体中修改unknownCode的值。

本例展示了值绑定的语法，但是其实没什么作用。标准的default分支就能得到同样的结果。把最后一个分支替换为default分支，如代码清单5-5所示。

代码清单5-5　用回default分支

```
...
switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "\(statusCode) is not a known error code."
}
```

代码清单5-4中的最后一个分支声明了一个常量，其值绑定为状态码的值。这意味着最后一个分支显然会匹配任何没有匹配上switch语句中前面分支的值，因此switch语句就被全覆盖了。

当删掉最后一个分支时，switch语句就没有被全覆盖了。这意味着我们需要为switch语句增加一个default分支。

5.2.3　where 子句

上面的代码都能运行，但是算不上很好。毕竟，状态码200其实不是错误——200表示成功！因此，switch语句最好不要匹配这些分支。

要修复这个问题，可以使用where子句来确保unknownCode不是代表成功的2xx。where能让

你额外检查一些条件，只有满足这些条件后才会匹配这个分支并绑定值。这个特性可以在switch中创建一些动态筛选条件，如代码清单5-6所示。

代码清单5-6　用where创建筛选条件

```
import Cocoa

var statusCode: Int = 404
var statusCode: Int = 204
var errorString: String = "The request failed with the error:"

switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode where (unknownCode >= 200 && unknownCode < 300)
                                     || unknownCode > 505:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "\(statusCode) is not a known error code."
    errorString = "Unexpected error encountered."
}
```

现在unknownCode分支指定了一个状态码范围，这意味着没有全覆盖。不过在这里没有问题，因为已经有default分支了。

只要不用Swift的fallthrough特性，switch语句就会在找到一个匹配分支并执行分支体代码后马上停止执行。当statusCode等于204时，switch会匹配第二个分支并将errorString设置为相应的值。所以，即使204处于where子句指定的区间内，switch语句也不会执行那个子句。

修改statusCode的值来练习where子句的用法，并确认其行为符合预期。

5.2.4　元组和模式匹配

有了statusCode和errorString后，把这两块信息拼接起来就很有用了。尽管两者逻辑相关，但是目前是存储在两个独立变量中的。元组（tuple）可以用来组合这两个变量。

元组是开发者认为具有逻辑关联的两个或多个值的有限组合。不同的值被组合为单个复合

值。组合的结果是一个元素的有序列表。

创建一个元组来组合statusCode和errorString，如代码清单5-7所示。

代码清单5-7　创建元组

```
import Cocoa

var statusCode: Int = 204
var statusCode: Int = 418
var errorString: String = "The request failed with the error:"

switch statusCode {
case 100, 101:
    errorString += " Informational, \(statusCode)."

case 204:
    errorString += " Successful but no content, 204."

case 300...307:
    errorString += " Redirection, \(statusCode)."

case 400...417:
    errorString += " Client error, \(statusCode)."

case 500...505:
    errorString += " Server error, \(statusCode)."

case let unknownCode where (unknownCode >= 200 && unknownCode < 300)
                                || unknownCode > 505:
    errorString = "\(unknownCode) is not a known error code."

default:
    errorString = "Unexpected error encountered."
}

let error = (statusCode, errorString)
```

将statusCode和errorString放进一对圆括号就可以创建元组，结果被赋给error常量。

元组的元素可以用索引访问。你可能已经注意到在运行结果侧边栏中显示的元组的值带着.0和.1，这就是元素的索引。输入代码清单5-8所示的代码可以访问元组内存储的每个元素。

代码清单5-8　访问元组的元素

```
...
let error = (statusCode, errorString)
error.0
error.1
```

你会看到运行结果侧边栏显示error.0（元组的第一个元素）和error.1（元组的第二个元素）分别是是418和"Unexpected error encountered."。

Swift的元组也可以保存命名元素。元组的命名元素能使代码更可读。要理解error.0和error.1

代表什么值有些困难，而利用命名元素就能更容易地读懂了，如error.code和error.error。

给元组的元素起能提供更多信息的名字，如代码清单5-9所示。

代码清单5-9 给元组的元素起名

```
...
let error = (statusCode, errorString)
error.0
error.1
let error = (code: statusCode, error: errorString)
error.code
error.error
```

现在可以通过与元素关联的名字访问元组元素了：statusCode的名字是code，errorString的名字是error。运行结果侧边栏会显示跟刚才一样的信息。

在switch语句的分支中使用区间其实就是模式匹配（pattern matching）的一个例子。这种形式的模式匹配被称为区间匹配（interval matching），因为每个分支都尝试匹配一个区间和给定值。元组在模式匹配中也很有用。

举个例子，想象有一个应用会发出多个Web请求。每次服务器的响应返回时，我们会保存HTTP状态码。然后，你想看是否有请求失败并且状态码是404（就是“请求的资源不存在”的错误）；如果有的话，查看是哪些请求。在switch语句中使用元组就能匹配非常特殊的模式。

参照代码清单5-10添加代码来创建并对元组进行匹配。

代码清单5-10 用元组做模式匹配

```
...
let error = (code: statusCode, error: errorString)
error.code
error.error

let firstErrorCode = 404
let secondErrorCode = 200
let errorCodes = (firstErrorCode, secondErrorCode)

switch errorCodes {
case (404, 404):
    print("No items found.")
case (404, _):
    print("First item not found.")
case (_, 404):
    print("Second item not found.")
default:
    print("All items found.")
}
```

这段代码先添加了几个新常量。firstErrorCode和secondErrorCode表示两个不同Web请求的HTTP状态码。errorCodes是组合这些状态码的元组。

新的switch语句会匹配几个分支来判断这些请求可能产生了什么样的404组合。第二个分支

和第三个分支的下划线（_）是能匹配任何值的通配符，这样能使这两个分支专注于某个特定请求的错误码。仅当两个请求都以错误码404失败时，才会匹配第一个分支。仅当第一个请求以404失败时，才会匹配第二个分支。仅当第二个请求以404失败时，才会匹配第三个分支。最后，如果都匹配不上，那就意味着没有请求是以状态码404失败的。

因为firstErrorCode确实是状态码404，所以运行结果侧边栏会显示"First item not found."。

5.3 switch 与 if/else

switch语句主要用于比较一个值和多个潜在匹配的分支。if/else语句则对于检查单个的条件更有用。switch还提供一系列强大的特性，比如说本章提到的一些特性，可以让你匹配区间、绑定值到本地常量或变量以及匹配元组中的模式。

有时候一个值可能与很多分支匹配，但是你只关心其中一种情况。这时你会忍不住使用switch语句。举个例子，想象你在检查一个Int类型的年龄常量来寻找特定年龄段的人口：18~35岁（也被称为“酷人群”，cool demographic）。你可能觉得写一个只有一个分支的switch语句是最好的选择，如代码清单5-11所示。

代码清单5-11　单个分支的switch

```
...
let age = 25
switch age {
case 18...35:
    print("Cool demographic")
default:
    break
}
```

age是设置为25的常量。age可能是0~100的任意数字，但是你只对某个特殊区间感兴趣。switch会检查age是否介于18和35之间。如果是，那么age就处于我们所需的人口年龄区间中，接着就可以执行一些代码。否则，age没有处于目标人口年龄区间中，那么default分支会匹配；它只是简单地用break控制转移语句把控制流转移到switch外面。

注意，这里必须有default分支；switch语句必须被全覆盖。如果你觉得这段代码不顺眼，那就对了。这里不需要做什么，所以用了break。不需要做什么的时候不写代码是最好的！

Swift提供了一种更好的方式实现这种逻辑。第3章中我们学习了if/else语句。Swift也有if-case语句来提供类似于switch语句的模式匹配能力，如代码清单5-12所示。

代码清单5-12　if-case

```
...
let age = 25
switch age {
case 18...35:
    print("Cool demographic")
default:
```

```
        break
    }

    if case 18...35 = age {
        print("Cool demographic")
    }
```

这种语法优雅多了，只要简单地检查age是否在给定区间内，而不需要写一个你并不关注的default分支。if-case的语法能让你关注关键的分支：age是否处于18~35的区间内。

if-case也可以像switch语句一样实现更复杂的模式匹配。比如，如果你想知道age是否大于等于21，就可以使用代码清单5-13。

代码清单5-13　带多个条件的if-case

```
...
let age = 25

if case 18...35 = age {
    print("Cool demographic")
}
if case 18...35 = age, age >= 21 {
   print("In cool demographic and of drinking age")
}
```

上面的新代码和前面的功能一样，但是有了些新的东西，逗号后面的代码会检查age是否大于等于21。在美国，这意味着此人到了可以喝酒的年龄。

if-case为只有一个分支的switch语句提供了优雅的替代品，而且使switch语句如此好用的模式匹配能力对if-case也是适用的。当你想用的switch语句只有一个分支，而且并不关心default分支时，就可以用if-case。因为if-case就是具备更强大模式匹配功能的if/else，所以也可以用通常的else块——但是这么做意味着其实写了default分支，也就没必要用if-case了。

5.4　青铜挑战练习

查看下面的switch语句，控制台会打印什么内容？有了答案以后，把代码输入playground来看看是否正确。

```
let point = (x: 1, y: 4)

switch point {
case let q1 where (point.x > 0) && (point.y > 0):
    print("\(q1) is in quadrant 1")

case let q2 where (point.x < 0) && point.y > 0:
    print("\(q2) is in quadrant 2")

case let q3 where (point.x < 0) && point.y < 0:
    print("\(q3) is in quadrant 3")
```

```
case let q4 where (point.x > 0) && point.y < 0:
    print("\(q4) is in quadrant 4")

case (_, 0):
    print("\(point) sits on the x-axis")

case (0, _):
    print("\(point) sits on the y-axis")

default:
    print("Case not covered.")
}
```

5.5 白银挑战练习

利用逗号分隔的列表可以为`if-else`添加更多的条件。比如说，可以检查一个人是否：a) 属于“酷人群”（cool demographic）；b) 在美国达到了可以饮酒的年龄；c) 不到30岁。为代码清单5-13添加一个条件判断来检查一个人的年龄是否符合以上条件。

第 6 章

循　　环

循环对于重复性的任务比较有用。它能重复执行一段代码，可以是重复指定的次数，也可以是在满足指定条件的情况下重复运行。使用循环可以避免出现冗长、重复的代码，所以要注意了！开发过程中会大量用到循环。

本章会介绍两种循环：

- for循环
- while循环

如果重复次数已知或者容易推断，用for循环对一个实例的指定元素或者容器中的实例进行循环是最理想不过的了。while循环则适合在满足某些条件时重复执行任务。两种循环都有变体。我们从for-in循环开始介绍，它会在指定的区间、序列或者容器的每个元素上执行一段代码。

6.1　for-in 循环

创建一个新的playground，命名为Loops。创建一个循环，如代码清单6-1所示。

代码清单6-1　for-in循环

```
import Cocoa

var str = "Hello, playground"

var myFirstInt: Int = 0

for i in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
}
```

首先声明一个类型为Int的变量myFirstInt，并初始化为0。接着，创建一个for-in循环。下面来看看该循环的组成部分。

for关键字意味着这是个循环。接着声明了一个迭代器（interator）i，用来表示循环的当前重复次数。迭代器是只在循环体内存在的常量，编译器会帮你管理这个常量。

在第一次循环中，其值是循环区间的第一个值。因为这里用...来创建1~5的闭区间，所以i

的第一个值就是1；在第二次循环中，i就是2；以此类推。你可以认为i在每次循环的开始都被一个新常量替换，其值为区间内下一个的值。

花括号（{}）里的代码会在每次循环时执行。每循环一次，myFirstInt就会增加1。增加myFirstInt后，下一行又出现了这个变量的名字，这是为了在运行结果侧边栏中显示其值。然后再将其值打印到控制台。增大和打印这两步会持续下去，直到i达到区间的结尾：5。这个循环可以用图6-1表示。

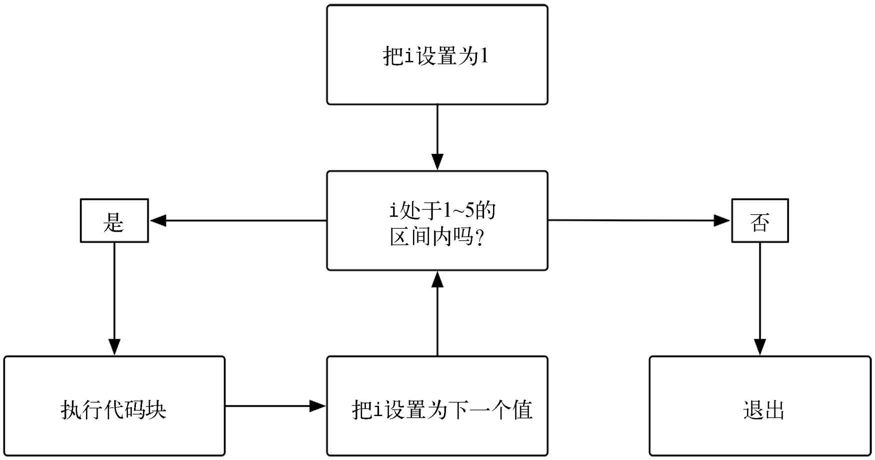


图6-1　在区间上循环

要看到循环的结果，在myFirstInt这行代码的运行结果侧边栏最右边找到并点击结果按钮，如图6-2所示。



图6-2　结果按钮

这个操作会打开结果视图（results view），它在playground的代码中内嵌展示实例的历史值。

通过点击并拖曳图像窗口的边缘可以将其扩大或缩小。

把鼠标指针移到新窗口内，你会看到可以选择图像上的单个点。举个例子，如果点击中间的点，playground会告诉你这个点的值是3（如图6-3所示）。

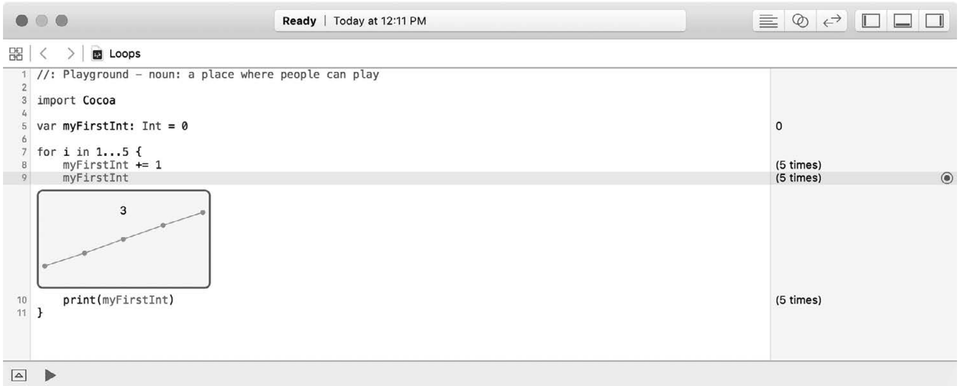


图6-3　选择图像上的一个值

因为i被声明为for-in循环的迭代器，所以能在循环的每次迭代过程中访问i。修改一下输出的代码来显示每次迭代过程中i的值，如代码清单6-2所示。

代码清单6-2　打印i不断变化的值到控制台

```
...
for i in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
    print("myFristInt equals \(myFirstInt) at iteration \(i)")
}
```

通过使用_可以忽略迭代器，这样可以不用显式地使用它。将命名常量替换为这一通配符，再把print()语句改回之前的实现，如代码清单6-3所示。

代码清单6-3　用_代替i

```
for i in 1...5 {
for _ in 1...5 {
    myFirstInt += 1
    myFirstInt
    print("myFirstInt equals \(myFirstInt) at iteration \(i)")
    print(myFirstInt)
}
```

for-in循环的实现确保某个特定的操作可以发生一定次数。它不会每循环一次就检查并报告迭代器的值。如果需要在循环体内引用迭代器的话，通常会用显式的迭代器i。

where

Swift的for-in循环支持where子句，类似于第5章那样。利用where子句可以更好地控制循环代码何时执行。利用where子句可以为执行循环代码所要满足的条件提供逻辑测试。如果where子句建立的条件没有得到满足，循环代码就不会运行。

举个例子，想象现在要写一个在区间上重复的循环，但是只有迭代器遇到3的倍数时才执行代码。用到的代码如代码清单6-4所示。

代码清单6-4　带where子句的for-in循环

```
...
for _ in 1...5 {
    myFirstInt += 1
    myFirstInt
    print(myFirstInt)
}

for i in 1...100 where i % 3 == 0 {
    print(i)
}
```

跟之前一样，创建局部常量i，然后就可以用在where子句的条件中。1~100区间内的每个整数都被绑定到i上。接着，where子句检查i能否被3整除。如果余数为0，循环会执行代码。结果是循环会打印1~100中所有3的倍数。

图6-4解释了这个循环的控制流。

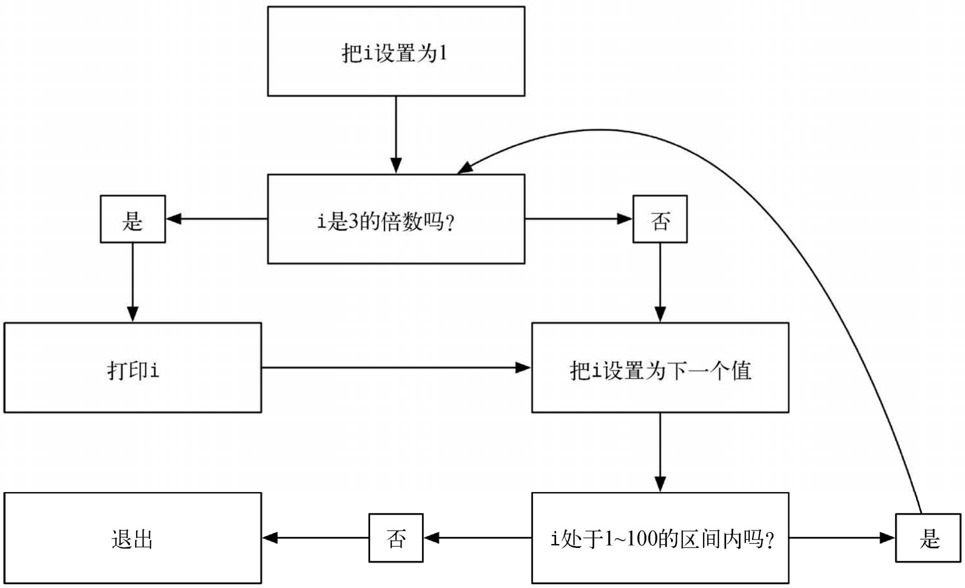


图6-4　带where子句的循环示意图

想象一下，如果没有where子句，要得到同样的结果该怎么做?

```
for i in 1...100 {
    if i % 3 == 0 {
        print(i)
    }
}
```

以上代码跟代码清单6-4中带where子句的循环做了同样的事情，但是就没那么优雅了。这里的代码行数更多，循环体内还嵌套着条件语句。一般来说，只要不是过于复杂、无法阅读，我们都偏好行数更少的代码。Swift的where子句可读性很好，所以我们通常选择这个更简洁的方案。

6.2　类型推断概述

看一下之前的一段代码：

```
for i in 1...5 {
    myFirstInt += 1
    print("myFirstInt equals \(myFirstInt) at iteration \(i)")
}
```

注意，这里没有把i声明为Int类型。它本应该是这样的：for i: Int in 1...5，但是没必要显式声明。i的类型可以用上下文推断出来。在本例中，因为指定的区间包含整数，所以可以推断i为Int类型。

类型推断很方便，能让你少敲些代码，从而少犯错。不过，有些情况下需要明确声明类型。出现这种情况的时候我们会强调。一般来说，我们建议尽量利用类型推断，本书中有很多这样的例子。

6.3　while 循环

只要条件为真，while循环会一直执行循环体内的代码。上面出现的for循环的例子大多可以用while循环改写。举个例子， 如代码清单6-5所示的while循环与代码清单6-1中的for循环相同。

代码清单6-5　while循环

```
...
var i = 1
while i < 6 {
    myFirstInt += 1
    print(myFirstInt)
    i += 1
}
```

图6-5展示了这段代码的执行流。

6

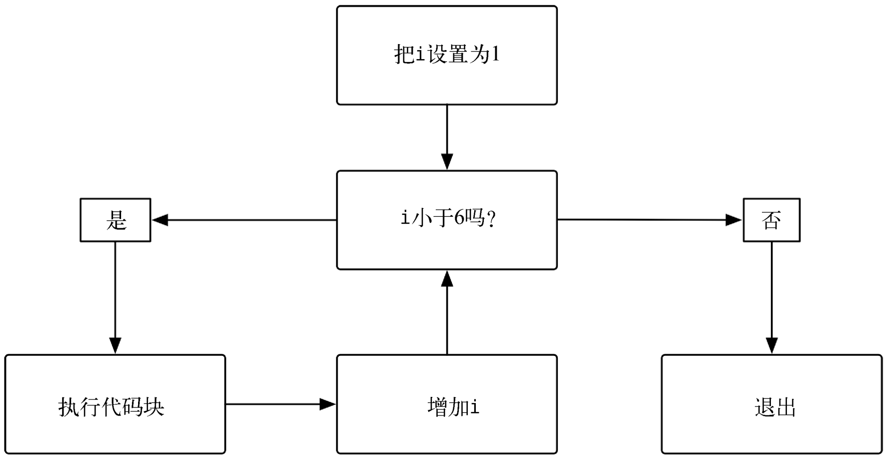


图6-5　while循环示意图

while循环初始化累加变量（var i = 1），计算条件（i < 6）；如果条件满足则执行代码（myFirstInt += 1, print(myFirstInt)，增加i），并回到while循环的开头判断循环是否应该继续。

i被声明为变量，因为我们计算的条件（i < 6）必须能改变。记住，只要条件为真，while循环就会一直运行。因此，while循环通常会检查在某个地方会发生变化的某个状态。否则，如果条件检查结果一直不变（或者说一直为真），那么while循环就会永远运行下去。永远不会结束的循环被称为死循环，一般来说属于bug。

while循环最适用于循环的重复次数未知的情况。比如，想象有一个简单的太空射击游戏，里面的太空船只要有护盾就持续使爆能枪（blaster）开火。一些外部因素可能会降低或提高飞船的护盾强度，所以实际的重复次数是未知的。不过只要护盾值大于0，爆能枪就应该一直开火。下面的代码片段说明了这个游戏的简单实现。

```
while shields > 0 {
    // 爆能枪开火!
    print("Fire blasters!")
}
```

6.4　repeat-while 循环

Swift还支持一种叫作repeat-while循环的while循环。repeat-while循环在其他语言中被称作do-while循环。while循环和repeat-while循环的区别在于何时计算条件。while循环在进入循环之前计算条件，这意味着while循环可能永远不会执行，因为其条件可能在第一次计算的时候就为假。repeat-while循环则至少执行一次，然后才计算条件。repeat-while循环的语法说明了这个区别。

```
repeat {
    // 爆能枪开火!
    print("Fire blasters!")
} while shields > 0
```

在这个repeat-while版本的太空射击游戏中，先执行包含print("Fire blasters!")的代码块，然后计算repeat-while的条件来判断循环是否应该继续。因此，repeat-while循环确保太空船至少用爆能枪开火一次。

repeat-while循环避免了有些令人沮丧的场景：如果因为某个反常的事故，太空船刚一建造完毕就失去了护盾，那会怎么样？可能护盾被迎面而来的小行星撞了个粉碎，爆能枪甚至来不及开一次火。这样的用户体验太差了。repeat-while循环确保爆能枪至少开火一次，从而避免这种虎头蛇尾的场景。

6.5 重提控制转移语句

现在在循环的语境中回顾一下控制转移语句。回想一下，第5章（用到了fallthrough和break）中的控制转移语句改变了通常的代码执行顺序。在循环语境中，你可以控制执行回到循环开头还是离开循环。

用太空射击游戏来阐述一下其工作原理。代码清单6-6使用continue控制循环语句来就地停止循环并从头开始。

代码清单6-6 使用continue

```
...
var shields = 5
var blastersOverheating = false
var blasterFireCount = 0
while shields > 0 {

    if blastersOverheating {
        print("Blasters are overheated! Cooldown initiated.")
        sleep(5)
        print("Blasters ready to fire")
        sleep(1)
        blastersOverheating = false
        blasterFireCount = 0
    }

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    // 爆能枪开火!
    print("Fire blasters!")

    blasterFireCount += 1
}
```

一下子多了好多代码，下面逐一分解。首先，我们增加了一些变量来记录太空船的某些信息：

- `shields`是`Int`类型，记录护盾强度，初始化为5；
- `blastersOverheating`是`Bool`类型，初始化为假，记录爆能枪是否需要冷却；
- `blasterFireCount`是`Int`类型，记录太空船开火的次数（用来判断爆能枪是否过热）。

创建变量后，我们写了两个`if`语句。它们都包含在一个`while`循环中，条件是`shields > 0`。第一个`if`语句检查爆能枪是否过热，第二个检查开火次数。对于第一个，如果爆能枪过热，会按步骤执行一段代码。打印信息到控制台，`sleep()`函数告诉系统等待5秒，模拟爆能枪的冷却周期。接着打印日志，说明爆能枪可以再次开火了，然后等待1秒（这样只是为了容易看到控制台接下去会打印什么），设置`blastersOverheating`为假，并重置`blasterFireCount`为0。

在护盾完整并且爆能枪冷却的情况下，太空船就准备好开火了。

第二个`if`语句检查`blasterFireCount`是否大于100。如果条件满足，就把`blastersOverheating`的布尔值置为真。从这里开始，爆能枪就过热了，需要返回循环的开头使其不再开火；用`continue`能做到这一点。因为太空船的爆能枪过热，所以第一个`if`语句计算为真，爆能枪关闭并冷却。

如果第二个条件计算为假，就像之前那样打印日志到控制台。接下来，把`blasterFireCount`增加1。增加变量后，循环会跳回开头并计算条件。它要么再重复一次，要么交出执行流、跳到紧接着循环右花括号的下一行。图6-6展示了这个执行流。

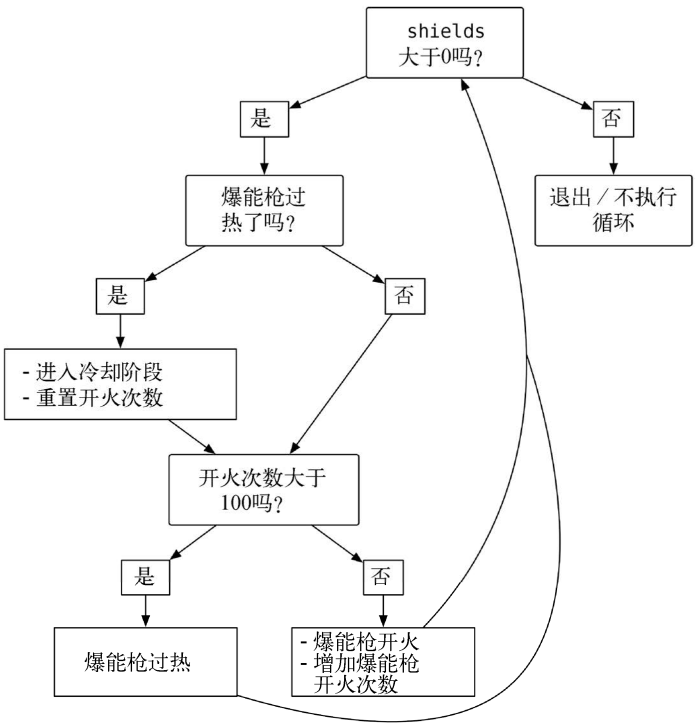


图6-6　`while`循环图示

注意，这段代码会无限执行。没有什么会改变shields的值，所以永远满足while shields > 0。如果什么都不变，电脑就有足够的电源永远运行，那么循环就会持续。我们称之为无限循环（infinite loop）。

但是任何游戏都会结束。假设用户摧毁500个太空恶魔后游戏结束。要退出循环，需要用到break控制转移语句，如代码清单6-7所示。

代码清单6-7　使用break

```
...
var shields = 5
var blastersOverheating = false
var blasterFireCount = 0
var spaceDemonsDestroyed = 0
while shields > 0 {

    if spaceDemonsDestroyed == 500 {
        print("You beat the game!")
        break
    }

    if blastersOverheating {
        print("Blasters are overheated! Cooldown initiated.")
        sleep(5)
        print("Blasters ready to fire")
        sleep(1)
        blastersOverheating = false
        blasterFireCount = 0
        continue
    }

    if blasterFireCount > 100 {
        blastersOverheating = true
        continue
    }
    // 爆能枪开火!
    print("Fire blasters!")

    blasterFireCount += 1
    spaceDemonsDestroyed += 1
}
```

这里增加了一个叫spaceDemonsDestroyed的新变量，会在每次爆能枪开火时增加。（很显然，你是个神射手。）接下来，增加一个新的if语句来检查spaceDemonsDestroyed变量是否等于500。如果是，打印胜利信息到控制台。

注意break的使用。break控制转移语句会退出while循环，并执行紧接着循环右花括号的代码。这样做是有道理的：如果用户摧毁500个太空恶魔，赢得了游戏，爆能枪就不需要再开火了。

6

6.6　白银挑战练习

*Fizz Buzz*是用来练习除法的游戏。用如下规则实现这个游戏：对于给定区间内的每个值，如果当前的数字能被3整除，就打印FIZZ。如果能被5整出，就打印BUZZ。如果能同时被3和5整除，就打印FIZZ BUZZ。如果既不能被3也不能被5整除，就直接打印这个数字。

举个例子，对于1~10的区间，玩*Fizz Buzz*游戏会得到这样的结果：1, 2, FIZZ, 4, BUZZ, FIZZ, 7, 8, FIZZ, BUZZ.

计算机喜欢玩*Fizz Buzz*。这个游戏很适合用循环和条件语句实现。在0~100的区间内进行循环，并正确地为区间内的每个数字打印FIZZ、BUZZ或者FIZZ BUZZ。

利用`if/else`条件语句和`switch`语句解决这个问题可以获得附加分。在使用`switch`时，要确保在各个分支中针对元组进行匹配。

第 7 章

字　符　串

在编程过程中，文本内容是用字符串表示的。你已经见到并用过字符串。比如，"Hello, playground"是一个字符串，出现在每个新建playground的开头。跟所有字符串一样，可以认为它是字符的有序集合。实际上，Swift字符串本身并不是集合，但是其底层内容确实以集合形式存在，而字符串类型提供了多种视角来一窥究竟。本章会更详细地介绍字符串的功能。

7.1　使用字符串

在Swift中，用String类型可以创建字符串。创建一个新的playground，命名为Strings，添加如代码清单7-1所示的String实例。

代码清单7-1　Hello, playground

```
import Cocoa

var str = "Hello, playground"
let playground = "Hello, playground"
```

上面的代码用字符串字面量创建了一个名为playground的String实例，这个字符串用引号把一段文本引起来了。

因为这个实例是用let关键字创建的，所以是常量。回想一下，常量意味着实例不能改变。如果你试图改变常量，编译器会报错。

现在创建一个字符串，但是这次的字符串实例是可变的，如代码清单7-2所示。

代码清单7-2　创建可变字符串

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
```

mutablePlayground是String的可变实例。也就是说，你可以改变其内容。用加法和赋值运算符在末尾加上标点，如代码清单7-3所示。

代码清单7-3　给可变字符串添加内容

```
...
let playground = "Hello, playground"
```

```
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"
```

看一下playground右边的运行结果侧边栏，你会看到实例变成了"Hello, mutable playground!"。

组成Swift字符串的字符都是Character类型。Swift的Character类型表示Unicode字符，组合起来形成String实例。

遍历一遍mutablePlayground字符串来实际看一下Character类型，如代码清单7-4所示。

代码清单7-4　mutablePlayground的Character实例

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}
```

这个循环在mutablePlayground的每个Character类型c上运行。循环访问了mutablePlayground字符串的characters属性。现在先别担心什么是属性，第16章会详细介绍这个话题。目前你只要知道属性是类型持有数据的一种方式就行了。在Swift中，用*点语法*（dot syntax）来访问属性，就像mutablePlayground.characters这样。

characters属性表示组成这个实例的字符集合。每循环一次会把字符串中的一个字符打印到控制台。每个字符会在控制台中单独打印一行，因为print()会在打印内容后换行。

输出看起来类似于图7-1。

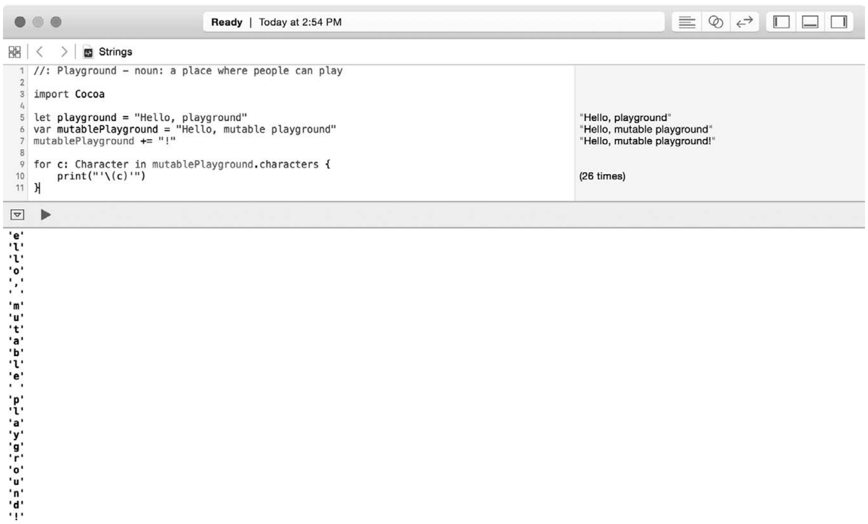


图7-1　打印字符串中的字符

7.2 Unicode

Unicode是字符编码的国际标准，目标是不用考虑平台即可无缝处理和表达字符。Unicode在计算机上表示人类语言（还有其他形式的通讯符号，比如emoji）。标准中的每个字符都被赋予了唯一的数。

Swift的`String`和`Character`类型构建于Unicode之上，并且把大部分的复杂细节都屏蔽了。不过，理解这两种类型如何使用Unicode还是很有用的。这些知识很可能会让你在将来节省不少时间、避免不少挫折。

7.2.1 Unicode 标量

从内部实现说，Swift字符串是由Unicode标量（Unicode scalar）组成的。Unicode标量是21位的数，表示Unicode标准中一个特定字符。比如，`U+0061`表示小写拉丁字母a。`U+1F60E`表示戴着墨镜的笑脸emoji。文本`U+1F60E`是书写Unicode字符的标准方式。1F60E部分是十六进制数。

创建一个常量，来看看在Swift和playground中如何使用特定的Unicode标量，如代码清单7-5所示。

代码清单7-5 使用Unicode标量

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
```

这次用了新语法创建字符串。引号我们已经很熟悉了，但是引号内部并不是一个之前见过的字符串字面量，跟侧边栏中的结果不一样，如图7-2所示。

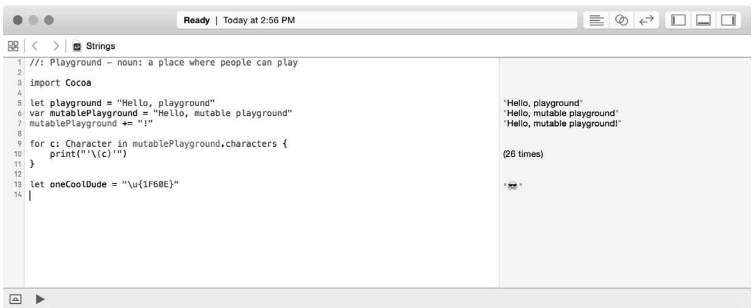


图7-2 侧边栏中的emoji

\u{}语法表示Unicode标量，十六进制数放在花括号中。在本例中，oneCoolDude被置为“戴墨镜”emoji的字符。

这跟我们熟悉的字符串有什么关系？其实Swift中每个字符串都由Unicode标量组成。那为什么看起来不熟悉呢？为了解释这一点，我们还需要讨论一些概念。

在Swift中，每个字符都由一个或多个Unicode标量构成。一个Unicode标量对应某种给定语言中的一个基本字符。我们之所以说字符是由“一个或多个”Unicode标量构成的，是因为还存在组合标量（combining scalar）。比如，U+0301表示可组合的重音符号（ ́）的Unicode标量。这个标量将重音符号放置在它前面的标量所对应的字符上面，也就是与前面的字符组合。用这个标量和小写拉丁字母a可以创建字符á，如代码清单7-6所示。

代码清单7-6　使用组合标量

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
let aAcute = "\u{0061}\u{0301}"
```

运行结果侧边栏会出现á，就是字母a和重音符号的组合。

Swift中的每个字符都是*扩展字形簇*（extended grapheme cluster）。扩展字形簇是人类可读的单个字符，由一个或多个Unicode标量组合而成。刚才我们把字符á拆成了两部分Unicode标量：字母和重音。把字符实现为扩展字形簇让Swift具备了处理复杂的脚本字符的灵活性。

Swift还提供了一种机制，能让我们看到字符串中所有的Unicode标量。比如，你可以利用unicodeScalars属性看到刚才创建的playground字符串实例中的所有Unicode标量，该属性持有所有Swift用来构建该字符串的标量。添加如代码清单7-7所示代码来查看playground的Unicode标量。

代码清单7-7　显示字符串背后的Unicode标量

```
...
let playground = "Hello, playground"
var mutablePlayground = "Hello, mutable playground"
mutablePlayground += "!"

for c: Character in mutablePlayground.characters {
    print("'\(c)'")
}

let oneCoolDude = "\u{1F60E}"
let aAcute = "\u{0061}\u{0301}"
```

```
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}
```

你会在控制台中看到如下输出：72 101 108 108 111 44 32 112 108 97 121 103 114 111 117 110 100。这些数是什么意思呢?

回忆一下unicodeScalars属性持有创建playground字符串实例所需的所有Unicode标量。控制台中的每个数对应一个字符串标量，表示字符串中的单个字符。但是这些数不是十六进制的Unicode数，而是用无符号32位整数表示的。第一个数72表示Unicode标量值为U+0048，即大写字母H。

7.2.2 标准等价

组合标量有其存在意义，不过Unicode也为某些常见字符提供了已经组合过的形式。比如，á有一个专属的标量，实际上不用分为字母和重音符号两部分。这个标量是U+00E1。创建一个新的字符串常量来使用这个Unicode标量，如代码清单7-8所示。

代码清单7-8 使用预组合字符

```
...
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"
```

如你所见，aAcutePrecomposed看起来和aAcute的值一样。实际上，如果检查两个字符是否相等，你会发现Swift确实认为它们相等，如代码清单7-9所示。

代码清单7-9 检查等价性

```
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

let b = (aAcute == aAcutePrecomposed) // 真
```

aAcute用两个Unicode标量创建，而aAcutePrecomposed只用了一个。为什么Swift认为两者等价呢？答案是标准等价（canonical equivalence）。

标准等价是指两个Unicode标量序列在语言学层面是否相等。对于两个字符或者两个字符串，如果它们具备相同的语言学含义和外观，那么无论是否用相同的Unicode标量创建，都认为两者相等。aAcute和aAcutePrecomposed是相等的字符串，因为两者都表示带上重音符号的小写拉丁字母a，而它们由不同的Unicode标量创建的事实并不影响这一点。

1. 计算元素数量

标准等价对字符串的元素数量计算会有影响。你可能会认为aAcute和aAcutePrecomposed的字符数量不同。下面用代码清单7-10中的代码来检查一下。

代码清单7-10　计算字符数量

```
...
let aAcute = "\u{0061}\u{0301}"
for scalar in playground.unicodeScalars {
    print("\(scalar.value) ")
}

let aAcutePrecomposed = "\u{00E1}"

let b = (aAcute == aAcutePrecomposed) // 真
print("aAcute: \(aAcute.characters.count);
       aAcutePrecomposed: \(aAcutePrecomposed.characters.count)")
```

使用characters的count属性来判断两个字符串的字符数量。count会遍历字符串的Unicode标量来判断其长度。运行结果侧边栏显示字符数量相等：都是1字符长。标准等价意味着无论是用组合标量还是预组合标量，结果都会被当成单个字符。

2. 索引和区间

由于可以把字符串理解为字符的有序集合，你可能会认为可以这样访问字符串中的某个字符：

```
let index = playground[3] // 'l'???
```

playground[3]使用了下标语法。在Swift中，方括号（[]）一般表示下标。下标可以在容器中获取特定的值。

3是一个*索引*，用来在容器中寻找特定元素。上面的代码在组成playground字符串的字符集合中选择第4个字符（第1个字符索引是0）。下面会详细介绍下标，第9章和第10章会介绍下标在数组和字典中的实际应用。

如果这样使用下标，会得到一个错误："'subscript' is unavailable: cannot subscript String with an Int." Swift编译器不允许用下标索引访问字符串中的特定字符。这个限制与Swift字符串和字符的存储方式有关。不能用整数作为索引访问字符串，因为Swift无法在不遍历前面每个字符的情况下知道指定的索引对应于哪个Unicode标量。这个操作很耗时。因此，Swift强迫你明确指定这个操作。

Swift用名为String.CharacterView.Index的类型记录索引。不用担心String.CharacterView.Index中的点（.），这只是说明Index是定义在CharacterView上的类型，而CharacterView又是定义在String上的类型。（第16章会详细介绍嵌套类型。）CharacterView类型负责以有序集合的形式提供字符串内部字符的访问接口。

如你所见，一个字符可能由多个Unicode码位组成。CharacterView的职责就是用一个Character对象表示每个码位，并且把这些字符拼接成正确的字符串。

在CharacterView上定义Index很方便。这样可以通过字符视图得到对字符串有意义的索引值。举个例子，要找出特定索引处的字符，首先用String类型的startIndex属性。这个属性会

以String.CharacterIndex.Index的形式返回字符串的起始索引。然后结合起始点和index(_:offsetBy:)方法往前移动直至到达选定的位置。(方法类似于函数，第12章会详细介绍。)

假设现在要知道本章开头创建的playgronud字符串的第5个字符，可以参考代码清单7-11。

代码清单7-11 找到第5个字符

```
...
let start = playground.startIndex
let end = playground.index(start, offsetBy: 4)
let fifthCharacter = playground[end] // "o"
```

使用字符串的startIndex属性来获取字符串的第一个索引。这个属性产生String.CharacterView.Index类型的实例。接着，使用index(_:offsetBy:)方法从起始点向前移动到你要的位置。offsetBy参数是Int类型，方法会把它加到第一个索引上。这里传入4表示第5个字符。

调用index(_:offsetBy:)的结果是String.CharacterView.Index，然后将其赋给end常量。最后，用end作为下标访问playground字符串，得到的结果是字符o。(记住，playground被设置为了"Hello, playground"。)

7

区间跟索引类似，都依赖String.CharacterView.Index类型。想象你要获取playground中的前5个字符，可以使用同样的start和end常量，如代码清单7-12所示。

代码清单7-12 获取区间

```
...
let start = playground.startIndex
let end = playground.index(start, offsetBy: 4)
let fifthCharacter = playground[end] // "o"
let range = start...end
let firstFive = playground[range] // "Hello"
```

start...end的结果被称为String.CharacterView.Index类型的闭区间，其工作方式与第6章的区间1...5类似。把新建的区间当作下标访问playground字符串。这个下标会把playground的前5个字符取出。结果是firstFive常量等于"Hello"。

7.3 青铜挑战练习

创建字符串常量empty并为它赋值空字符串：let empty = ""。判断字符串是否包含字符是很有用的。用empty的startIndex和endIndex属性来判断字符串是否真的为空。接着，通过Xcode的Help菜单查阅文档来完成这项检查。

7.4 白银挑战练习

把"Hello"字符串替换为从对应的Unicode标量创建的实例。可以在网上找到合适的代码。

第 8 章 可空类型

可空类型（optional）是Swift的独特特性，用来指定某个实例可能没有值。看到可空类型时，你会知道该实例一定：要么有值并且已经可用，要么没有值。如果一个实例没有值，就称其为nil。

任何类型都可以用可空类型来说明一个实例可能是nil。这个特性将Swift和Objective-C区分开来，后者只允许对象是nil。

本章讲述如何声明可空类型，如何使用可空实例绑定（optional binding）来检查某个可空实例是否为nil并且在有值的情况下使用其值，以及如何使用可空链式调用（optional chaining）来查询一连串可空值。

8.1 可空类型

Swift的可空类型让这门语言更加安全。一个可能为nil的实例应该被声明为可空类型。这意味着如果一个实例没有被声明为可空类型，它就不可能是nil。通过这种方式，编译器知道一个实例是否可能为nil。这种显式声明可以让代码更具表达能力，也更安全。

现在来看一下如何声明可空类型。创建一个新的playground并命名为Optionals。输入如代码清单8-1所示的代码片段。

代码清单8-1　声明可空类型

```
import Cocoa

var str = "Hello, playground"

var errorCodeString: String?
errorCodeString = "404"
```

首先声明一个名为errorCodeString的变量，以字符串的形式来持有错误码信息。接着，显式声明errorCodeString的类型是String；跟之前的形式略有不同，这次在String后面加上了?。?使得errorCodeString成为了可空的String类型。

声明了可空实例并为其赋值后，就可以参照代码清单8-2将其值打印到控制台了。

代码清单8-2　打印可空实例的值到控制台

```
import Cocoa
```

```
var errorCodeString: String?
errorCodeString = "404"
print(errorCodeString)
```

打印errorCodeString的值会显示Optional("404")。如果没有给errorCodeString赋值会怎么样？试试看！如代码清单8-3所示，注释掉为errorCodeString赋值的那行代码。

代码清单8-3　打印可空实例的nil值到控制台

```
import Cocoa

var errorCodeString: String?
// errorCodeString = "404"
print(errorCodeString)
```

查看控制台，你会看到打印的值是nil。

但是打印nil到控制台没什么用。你希望知道的是变量何时为nil，以便相应地执行一些代码。在这种情况下，可以使用条件语句来针对变量的值做到这一点。

比如，如果某个操作产生了错误，就把错误赋给一个新变量并打印到控制台。添加如代码清单8-4所示代码到playground。

代码清单8-4　增加条件语句

```
import Cocoa

var errorCodeString: String?
// errorCodeString = "404"
print(errorCodeString)
if errorCodeString != nil {
    let theError = errorCodeString!
    print(theError)
}
```


让我们看一下上面的代码做了什么。这段代码增加了一个条件语句，如果errorCodeString不是nil就会执行相应的代码（回忆一下，!=就是“不等于”的意思）。

我们在条件体中创建了一个叫作theError的新常量来持有errorCodeString的值。要做到这一点，需要在errorCodeString后面增加!。在这里，感叹号的作用是**强制展开**（forced unwrapping）。

强制展开会访问可空实例封装的值，这样就能把"404"取出并赋给常量theError。之所以称之为“强制”展开，是因为无论是否有值，都会访问封装的值。也就是说，!假设有这样一个值；如果没有，这样展开会产生运行时错误。

强制展开具有一定的危险性。如果可空实例没有值，程序会在运行时触发陷阱。因为本例先检查并确保了errorCodeString不是nil，所以强制展开并不危险。尽管如此，我们还是建议谨慎和节制地使用强制展开。

最后，把新变量的值打印到控制台。

如果不强制展开errorCodingString的值而是直接把可空实例赋给常量theError会怎么样？theError的值还是会正确地输出到控制台。既然这样，那为什么还要展开可空实例的值并赋给常量呢？要回答这个问题，需要更好地理解可空类型。

如果省略errorCodeString末尾的感叹号，那就只是把可空的String而不是把实际错误码的String值赋给常量。事实上，errorCodeString的类型是String?。String?和String不是相同的类型——如果你有一个String变量，就无法在不展开可空实例的情况下将String?的值赋给这个变量。

可空的errorCodeString是nil，因为声明时并没有为其赋值。下一行代码为其赋值"404"。通过比较可空实例的值和nil可以判断它是否有值。上面的代码检查了errorCodeString是否有值。如果其值不等于nil，展开errorCodeString就是安全的。

在条件语句内部创建常量有点笨重。幸运的是，有更好的办法有条件地把可空实例的值绑定给常量。这种方法称为可空实例绑定。

8.2　可空实例绑定

可空实例绑定（optional binding）是一种固定模式，对于判断可空实例是否有值很有用。如果有值，就将其赋给一个临时常量或变量，并且使这个常量或变量在条件语句的第一个分支代码中可用。这样可以让代码更简洁，同时保持表达力。下面是基本的语法：

```
if let temporaryConstant = anOptional {
    // 用temporaryConstant做一些事情
} else {
    // anOptional没有值，也就是说anOptional为nil
}
```

有了这种语法，就可以利用可空实例绑定对上例进行重构了，如代码清单8-5所示。

代码清单8-5　可空实例绑定

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if errorCodeString != nil {
    let theError = errorCodeString!
if let theError = errorCodeString {
    print(theError)
}
```

正如你所见，可空实例绑定的语法与在条件语句中创建常量基本一致。theError常量从条件语句体中移到了第一行，让theError变成了在条件语句的第一个分支中可用的临时常量。换句话说，如果可空实例有值，那么就会有一个临时常量；如果条件计算为真，其执行的代码块就可以使用这个常量。

此外，不需要再强制展开可空实例了。如果转换成功，那么这个操作已经自动完成了，可空

实例的值已经在你声明的临时常量中可用了。最后要注意，如果需要在条件语句的第一个分支中修改theError，那么可以用var关键字声明它。

假设你想把errorCodeString转换成相应的整数形式，可以用嵌套的if let绑定来完成任务，如代码清单8-6所示。

代码清单8-6　嵌套的可空实例绑定

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString {
    print(theError)
    if let errorCodeInteger = Int(theError) {
        print("\(theError): \(errorCodeInteger)")
    }
}
```

注意，第二个if let在第一个里面，这样可以让theError在第二个可空实例绑定中可用。这里还用到了在第4章出现过的语法来转换整数。

上例的Int(theError)可以把theError变量中的String实例转换为对应的Int。这个操作可能会失败，比如字符串"Error!"本来就无法转换为整数。因此，Int(theError)返回一个可空类型，以防无法找到与给定字符串对应的Int。

在第二个绑定中，Int(theError)的结果被展开并赋给errorCodeInteger，使得整数值也可用了。然后就可以在print()的调用中使用这两个新常量来打印信息到控制台。

嵌套可空实例绑定可能会显得错综复杂。如果只是几个可空实例，问题不会太大；不过可以想象一下，如果再多几个需要展开的可空实例，这种嵌套会变得多复杂。我们可以把可空实例绑定嵌套得很深，形成“末日金字塔”（Pyramid of Doom，指很多的缩进层次）。值得庆幸的是，单个if let绑定就可以展开多个可空实例，如代码清单8-7所示。这个特性对于避免嵌套多个if let很有帮助，比如代码清单8-6这样令人难受的代码（以及更糟的代码）。

代码清单8-7　展开多个可空实例

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString, let errorCodeInteger = Int(theError) {
    if let errorCodeInteger = Int(theError) {
    print("\(theError): \(errorCodeInteger)")
    }
}
```

现在一行代码展开了两个可空实例：if let theError = errorCodeString和let errorCodeInteger = Int(theError)。首先展开errorCodeString，其值被赋给theError。我们还用到了Int(theError)尝试把theError转换为Int。因为结果是可空类型，所以需要展开

并将其值绑定到errorCodeInteger。如果两个绑定中有任何一个返回nil，那么条件语句的成功代码块就不会执行。不过在本例中，errorCodeString有值且theError能成功展开，因为theError能转换为整数。

可空实例绑定还能进行额外的检查，写法跟前面出现过的标准if语句差不多。假设当错误码的值为404时你才对其感兴趣，如代码清单8-8所示。

代码清单8-8　可空实例绑定和额外的检查

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
if let theError = errorCodeString, let errorCodeInteger = Int(theError) ,
    errorCodeInteger == 404 {
    print("\(theError): \(errorCodeInteger)")
}
```

（不要漏掉这段代码所添加的逗号。）

现在，如果errorCodeInteger等于404，条件语句就计算为真。只有当两个可空实例都成功展开时，才会执行最后一个子句（errorCodeInteger == 404）。因为theError是"404"，而这个字符串可以转换为整数404，所以所有条件都能满足，404: 404会被打印到控制台。

8.3　隐式展开可空类型

现在值得讲一下*隐式展开可空类型*（implicitly unwrapped optional），不过在后面讨论类和类初始化之后才会正式用到它。隐式展开可空类型与普通可空类型类似，只是有一个重要的区别：它们不需要展开。怎么会这样？这跟声明方式有关。看一下如下代码，这段代码用隐式展开可空类型重构了上面的例子。

```
import Cocoa

var errorCodeString: String!
errorCodeString = "404"
print(errorCodeString)
```

这里的可空类型是用!声明的，表示这是一个隐式展开可空类型。条件语句被删除了，因为使用隐式展开可空类型意味着我们对于其有值要比使用普通可空类型更有信心。确实，隐式展开可空类型的强大和灵活性跟不必展开就能访问其值有关。

不过要注意，这种强大和灵活性也伴随着一定的危险性：如果隐式展开可空实例没有值的话，访问其值会导致运行时错误。为此，我们建议只要某个实例有可能是nil，就别用隐式展开可空类型。实际上，因为隐式展开可空类型安全性较差，所以如果你不是明确指出想要使用隐式展开可空类型，Swift就会给你普通的可空类型。

我们再回顾一下上面这个例子来看看实际效果。假设errorCodeString被置为nil。如果再声明一个字符串类型的常量anotherErrorCodeString并试图把errorCodeString的内容（可能

为空）赋给它会怎么样呢？如果要把errorCodeString赋给另外一个实例而又没有显式声明类型，你觉得Swift会怎么推断新实例的类型呢？

```
import Cocoa

var errorCodeString: String! = nil
errorCodeString = "404"
print(errorCodeString)

let anotherErrorCodeString: String = errorCodeString // 能工作吗？
let yetAnotherErrorCodeString = errorCodeString // 是可空实例还是隐式展开可空实例呢？
```

第一个问题的答案是会触发陷阱。如果errorCodeString为空，把它的值赋给字符串类型的anotherErrorCodeString会导致运行时错误。为什么？因为anotherErrorCodeString显式声明了类型，不可能是可空实例。

对于第二个问题，Swift会尽可能推断最安全的类型：普通的可空实例。yetAnotherError-CodeString的类型会是String?，值为nil。要访问其值，必须展开可空实例，这样有助于增加代码的安全性。这个特性让类型推断在默认情况下是安全的，从而阻止不安全代码的扩散。

如果想让yetAnotherErrorCodeString是隐式展开可空类型，那么编译器要求进行显式的声明。要像let yetAnotherErrorCodeString: String! = errorCodeString这样把想要的实例声明为隐式展开可空类型。

最好只在一些特殊情况下使用隐式展开可空类型。正如我们指出的，它主要的应用场景是类初始化，第17章会详细讨论。目前，你已经知道了关于隐式展开可空类型的基础知识，实际遇到也能理解究竟是怎么回事了。

8

8.4 可空链式调用

与可空实例绑定类似，*可空链式调用*（optional chaining）提供了一种对可空实例进行查询以判断其是否包含值的机制。两者的一个重要区别是，可空链式调用允许程序员把多个查询串联为一个可空实例的值。如果链式调用中的每个可空实例都包含值，那么每个调用都会成功，整个查询链会返回期望类型的可空实例。如果查询链中的任意可空实例是nil，那么整个链式调用会返回nil。

让我们从一个简洁的例子开始。假设应用因为某种原因有一个自定义错误码。如果遇到404，就用自定义错误码以及展示给用户的错误描述代替。在playground中增加如代码清单8-9所示的代码。

代码清单8-9 可空链式调用

```
import Cocoa

var errorCodeString: String?
errorCodeString = "404"
var errorDescription: String?
```

```
if let theError = errorCodeString, let errorCodeInteger = Int(theError),
    errorCodeInteger == 404 {
    print("\(theError): \(errorCodeInteger)")
    errorDescription = "\(errorCodeInteger + 200): resource was not found."
}

var upCaseErrorDescription = errorDescription?.uppercased()
errorDescription
```

上面的代码增加了一个名为errorDescription的新变量。在if let的成功分支块，我们又创建了字符串插值并将其赋给errorDescription。在创建字符串插值时，我们利用\(errorCodeInteger + 200)对404进行增加，得到自定义错误码604（可以是任意一个数字，它在理论上对于这个应用来说是唯一的）。最后，增加一些关于这个错误的额外信息。

接着，用可空链式调用创建一个新的错误描述实例，使用全大写文本可能是为了说明其紧迫性。这个实例名为upCaseErrorDescription。errorDescription末尾的问号表示这行代码开始了可空链式调用的过程。如果errorDescription没有值，就没有字符串需要被转换成大写。这样，upCaseErrorDescription就是nil。这一点表明可空链式调用会返回可空实例。

因为errorDescription有值，所以描述信息会被转换成大写并再次将这一新值赋给upCaseErrorDescription。运行结果侧边栏应该会显示更新过的值："604: THE RESOURCE WAS NOT FOUND."。

8.5 原地修改可空实例

可空实例可以被“原地”修改，从而免去创建新变量或常量的麻烦。给upCaseErrorDescription增加一个append(_:)调用（如代码清单8-10所示）。

代码清单8-10 原地修改

```
...
upCaseErrorDescription?.append(" PLEASE TRY AGAIN.")
upCaseErrorDescription
```

原地修改可空实例非常有用。在本例中，要做的只是更新可空实例中的字符串，不需要任何返回值。如果可空实例有值，就给字符串增加一些文本；如果没有，就什么都不做。

这就是原地修改可空实例所做的事情。到目前为止，upCaseErrorDescription末尾的?与可空链式调用的作用差不多：如果有值就将其暴露给我们。如果upCaseErrorDescription为nil，那么可空实例就不会被修改，因为没有值需要更新。

值得一提的是，上面的代码也可以使用!操作符。这个操作符会强制展开可空实例——你已经知道了这个操作可能很危险。如果upCaseErrorDescription为nil，那么upCaseErrorDescription!.append(" PLEASE TRY AGAIN.")会导致运行时崩溃。

正如上面讲到的，大多数时候最好用?。只有在你知道可空实例不会为nil或者一旦可空实例是nil那么唯一合理的动作就是崩溃时才使用!操作符。

8.6 nil 合并运算符

处理可空类型时的一个常见操作是：要么获取其值（如果可空实例有值），要么使用某个默认值（如果可空实例是nil）。比如，从errorDescription中取出错误信息时，如果字符串并没有包含错误，那么你可能希望默认信息是"No error"。可以用可空实例绑定完成这个任务，如代码清单8-11所示。

代码清单8-11 用可空实例绑定解析errorDescription

```
...
let description: String
if let errorDescription = errorDescription {
    description = errorDescription
} else {
    description = "No error"
}
```

这样写有个问题，那就是需要写很多代码来完成一个本应该很简单的操作：从可空实例中获取值或者使用"No error"（如果可空实例为nil的话）。可以用nil合并运算符（nil coalescing operator）??来达到这个目的。在代码清单8-12中看看怎么使用它。

代码清单8-12 使用nil合并运算符

```
...
let description: String
if let errorDescription = errorDescription {
    description = errorDescription
} else {
    description = "No error"
}

let description = errorDescription ?? "No error"
```

??的左边必须为可空实例；在本例中是errorDescription，一个可空的String。右边必须是非可空的同类型实例；在本例中是"No error"，就是String类型。如果左边的可空实例是nil，那么??会返回右边的值。如果左边的可空实例不是nil，那么??会返回可空实例中包含的值。

试试修改errorDescription让其不包含错误。看一下description得到的值是否为"No error"，如代码清单8-13所示。

代码清单8-13 修改errorDescription

```
...
errorDescription = nil
let description = errorDescription ?? "No error"
```

本章涉及的内容繁多，你学到了很多新知识。无论你的开发水平如何，可空类型都是一门新知识。这是Swift的一个强大特性。

作为开发者，你经常需要在实例中表达nil。可空类型能让你追踪这些实例是否为nil，并

8

提供一种机制来进行适当的响应。

如果还不是很适应可空类型也别担心，你会在后面的章节中经常接触到它。

8.7　青铜挑战练习

可空类型最好用来表示本来就可以为空的概念，即适合用来表示缺失某些东西的场景。不过缺失不等同于零。举个例子，如果写代码为银行账户建模，而用户的给定账户没有余额，那么值为`0`比`nil`更合理。换句话说，用户并不是没有账户，只是没有钱而已。看看下面的例子，选择合适的类型。

- 一个人拥有的孩子数：`Int`还是`Int?`
- 一个人饲养的宠物的名字：`String`还是`String?`

8.8　白银挑战练习

本章开头提到，当可空实例为`nil`时访问其值会导致运行时错误。在可空实例为`nil`时用强制展开来人为制造这个错误，然后研究一下这个错误，理解这个错误告诉了你什么信息。

Part 3

第三部分

容器和函数

作为程序员，你经常需要将一组相关的值存放在一起。Swift的容器对象能帮你做到这一点，第三部分就会介绍Swift的各种容器类型。

Swift提供函数来帮助开发者将数据转换为对用户有意义的东西或者执行一些其他任务。这一部分的几章将描述如何用Swift语言提供的系统函数和我们创建的自定义函数来达成目标。

第 9 章

数　　组

9

编程中的一个重要任务是把逻辑相关的一组值放在一起。比如，想象你的应用要保存用户的好友列表、最爱的图书、旅行地点等。通常有必要具备将这些值放在一起并在代码中传递的能力。容器类型让这些操作变得方便。

Swift有一组容器类型，首先介绍的是*数组*（array）。

数组是值的有序集合。数组的每个位置都用索引标记，任何值都可以在数组中出现多次。数组通常用于值的顺序很重要或者很有用的场合，但是值的顺序是否有意义并不是先决条件。不像Objective-C，Swift的`Array`类型可以持有任何类型的值——对象和非对象都可以。

开始之前，先创建一个新的Swift playground，命名为Arrays。

9.1　创建数组

在本章中，你会创建一个表示目标清单（未来想做的事情）的数组。首先声明一个数组，如代码清单9-1所示。

代码清单9-1　创建数组

```
import Cocoa

var str = "Hello, playground"

var bucketList: Array<String>
```

这里创建了一个名为`bucketList`的变量，类型是`Array`。大部分语法看起来应该都挺熟悉。比如，关键字`var`表示`bucketList`是变量。这意味着`bucketList`可变，所以我们可以修改这个数组。不可变数组同样存在，我们会在本章稍后讨论。

语法中的新东西是`<String>`。这句代码告诉`bucketList`它应该接受什么样的实例。在这里，数组会接受`String`类型的实例。数组可以持有任何类型的实例。因为这个数组会持有与未来目标相关的信息，所以用`String`实例是合理的。

还有另一个语法可以声明数组。在playground中进行修改，如代码清单9-2所示。

代码清单9-2　换一种语法

```
import Cocoa

var bucketList: Array<String>
var bucketList: [String]
```

这里，方括号表示bucketList是Array实例，而String表示bucketList接受什么类型的值。

现在只是声明了bucketList，还没有初始化。这意味着它还没有准备好接受String类型的实例。如果你现在试着添加目标到bucketList中，会得到一个错误，说你在数组还未初始化bucketList时就添加实例。

修改声明bucketList的代码来同时初始化数组，如代码清单9-3所示。

代码清单9-3　初始化数组

```
import Cocoa

var bucketList: [String] = ["Climb Mt. Everest"]
```

这里用到了赋值运算符=和数组字面量语法["Climb Mt. Everest"]。数组字面量是用任意其包含的实例初始化数组的快捷语法。本例用攀登珠穆朗玛峰的目标初始化了bucketList。

像其他类型一样，数组可以利用Swift的类型推断能力来声明。删除类型声明来使用类型推断，如代码清单9-4所示。

代码清单9-4　使用类型推断

```
import Cocoa

var bucketList: [String]= ["Climb Mt. Everest"]
```

实际上没什么变化：bucketList还是包含同样的目标，仍然只接受String类型的实例。唯一的区别就是，现在是基于初始化用到的实例类型来推断出这个信息。如果你试图把整数添加到数组中，会看到一个不能把Int实例添加到其中的错误，因为数组期望的是String类型的实例。

现在知道了如何创建和初始化数组，是时候学习如何访问和修改数组元素了。

9.2　访问和修改数组

现在有一个目标清单了？不错啊！遗憾的是，你的愿望还没有完全添加进去。你是一个有趣的人，对生活充满热情，因此再给bucketList添加一些值吧。用另一个目标来更新清单，如代码清单9-5所示。

代码清单9-5　热气球冒险

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
```

这里用到append(_:)给bucketList添加值。append(_:)方法接受一个参数，类型可以是

数组接受的任何类型，并将其变为数组的新元素。

playground看起来应该如图9-1所示。



图9-1　添加目标到bucketList

用append(_:)函数再给bucketList添加一些未来的冒险，如代码清单9-6所示。

代码清单9-6　这么多愿望!

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
```

现在bucketList中有6个目标了。不过如果你改变心意了该怎么办？也可以想得积极一点，如果你完成了清单中的某个目标该怎么办?

假设上个周末你把自己安排得舒舒服服的，花10个小时看完了《魔戒》（*Lord of the Rings*）系列电影。那现在就是时候把这个目标从清单里拿掉了。用函数remove(at:)来删除，如代码清单9-7所示。（数组是从零开始索引的，所以"Climb Mt. Everest"位于索引0而"Watch the Lord of the Rings trilogy in one day"位于索引2。）

代码清单9-7　从数组中删除元素

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.remove(at: 2)
bucketList
```

为了确认bucketList中第二个索引的值已经被删除，在运行结果侧边栏中高亮最后一行（bucketList），然后点击像眼睛的那个按钮。这个功能称作快速查看（参见图9-2）。在快速查看窗口中往下滚动就能看到数组中元素的个数现在是5。之前在第二个索引的元素“电影马拉松”消失了。"Go on a walkabout"现在占据了第二个索引。

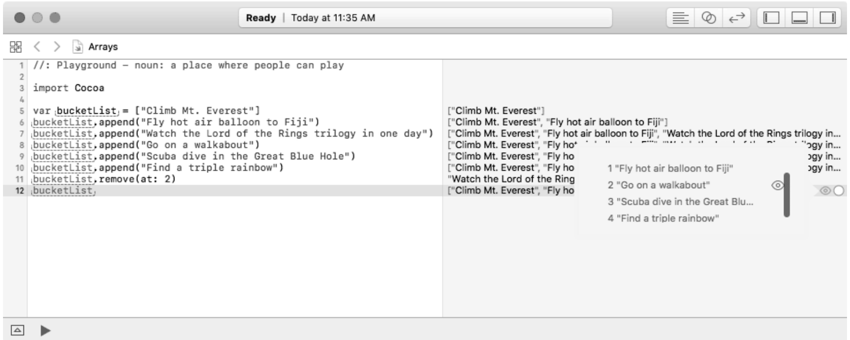


图9-2 试试快速查看

上周末的时间花在“电影马拉松”上之后，你决定本周末出去转转。你在参加聚会时和人们谈起了目标清单。当你说起你宏伟的目标时，众人目瞪口呆。有人倒吸一口气问：“你到底有多少目标要实现？”

没问题！要找出数组中的元素个数很容易。数组会用count属性记录其中的元素。用这个属性来打印清单中的目标数到控制台，如代码清单9-8所示。

代码清单9-8 获取数组元素个数

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.remove(at: 2)
bucketList
print(bucketList.count)
```

“5个，”人们惊叹，“那可要做很多事情啊！”聚会快结束了，大家都要回家好好想想自己的目标清单，他们恳请你告诉他们你的前三个目标。如代码清单9-9所示，这用下标（subscripting）很容易实现。下标能让你访问数组在指定索引处的值。

代码清单9-9 用下标寻找前三个目标

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
```

```
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.remove(at: 2)
bucketList
print(bucketList.count)
print(bucketList[0...2])
```

前面我们已经用过下标的方括号语法（`[0...2]`）。注意，前三个元素被打印到了控制台。（用同样的基本语法可以打印单个元素，比如`print(bucketList[2])`。）

下标是一个强大的特性。假设在谈话过程中有人问你："你打算去哪里进行丛林流浪[1]呢？"这个问题让你意识到这个目标还需要更明确一些。毕竟，不是去什么地方都行的，只有去澳大利亚才是丛林流浪。可以用下标来修改指定索引的元素（或者索引区间），如代码清单9-10所示。

代码清单9-10　用下标添加信息

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.remove(at: 2)
bucketList
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList
```

这里使用加法和赋值运算符+=来给索引2的元素增加文本。之所以能这样赋值，是因为索引2的实例和你添加的实例是相同的类型——`"Go on a walkabout"`和`" in Australia"`都是`String`类型。因此，索引2的值被修改为：`"Go on a walkabout in Australia"`。

这些冒险让你很兴奋，结果睡不着了。读书通常有助于入睡，所以你开始读关于攀登珠穆朗玛峰的书。你发现这很危险，决定把第一个目标换成不那么宏伟的目标（攀登乞力马扎罗山），如代码清单9-11所示。

代码清单9-11　替换数组元素

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
```

① 英文原文是walkabout，指澳洲土著的定期丛林漫游活动。——编者注

```
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

看！现在好多了。现在第一个目标安全了一点，不过仍然富于冒险精神。

你现在对目标清单的内容很满意，但是对于要输入5次bucketList.append不太满意。你对自己说："应该还有更好的办法！"

然后你想到了什么："我知道如何使用循环！如果用所有要添加的目标创建一个数组会怎么样？我可以遍历这个数组，然后在每次循环时使用append(_:)。这样只要写一次bucketList.append就行了！"

就是这么做（参见代码清单9-12）。

代码清单9-12　用循环从一个数组添加元素到另一个数组

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
bucketList.append("Fly hot air balloon to Fiji")
bucketList.append("Watch the Lord of the Rings trilogy in one day")
bucketList.append("Go on a walkabout")
bucketList.append("Scuba dive in the Great Blue Hole")
bucketList.append("Find a triple rainbow")
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]

for item in newItems {
    bucketList.append(item)
}
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

首先创建一个待添加目标的数组，名为newItmes。接着创建一个for-in循环来遍历数组的每个元素并将其添加到bucketList。在循环的本地作用域中用item变量来添加元素到bucketList数组。

对代码的重构使其更简洁，同时保持了表达力，你感到很满意。正在昏昏欲睡之时，你突然灵光一现。

“现在这样很好，”你想，“但是我可以做得更好。也许我可以用加法和赋值运算符！”确实可以。可以像用+=把一个整数加到另一个整数上一样把一个数组加到另一个数组上，如代码清单9-13所示。

代码清单9-13 用加法和赋值运算符重构代码

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]

for item in newItems {
    bucketList.append(item)
}
bucketList += newItems
bucketList
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList
```

+=运算符是把新元素的数组添加到现存目标清单的好办法。

最后，假设你有了一个新目标——坐雪橇穿越阿拉斯加。这个目标要比澳大利亚丛林流浪更重要，但是比不上坐热气球到斐济。用`insert(_:at:)`函数来把新元素添加到数组的指定索引位置，如代码清单9-14所示。

代码清单9-14 插入新目标

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
bucketList += newItems
bucketList
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
```

```
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", at: 2)
bucketList
```

`insert(_:at:)`函数有两个参数。第一个参数接受要添加到数组的实例。（回忆一下，这个数组接受`String`实例。）第二个参数接受你想添加的元素在数组中位置的索引。

现在清单圆满完成，你睡着了，梦见自己正坐着热气球飞往斐济的群岛。

9.3　数组相等

第二天早上，你醒来之后去隔壁咖啡厅。在那里遇到了一个名叫Myron的朋友，他也跟你一起参加了聚会。Myron被你的`bucketList`启发，决定模仿你定一个自己的目标清单。他在聚会结束后回到家把你的目标都写了下来，现在想确认一切无误。

在跟Myron同步了聚会后的改变之后，就要对比你们的目标清单数组元素以确保两者是一样的。用==检查是否相等，如代码清单9-15所示。

代码清单9-15　检查两个数组是否相等

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
bucketList += newItems
bucketList
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", at: 2)
bucketList

var myronsList = [
                  "Climb Mt. Kilimanjaro",
                  "Fly hot air balloon to Fiji",
                  "Toboggan across Alaska",
                  "Go on a walkabout in Australia",
                  "Find a triple rainbow",
                  "Scuba dive in the Great Blue Hole"
                  ]

let equal = (bucketList == myronsList)
```

既然两个数组的内容都是一样的，你可能会期望`equal`是真。不过`equal`却被判定为假。为什么？

记住，数组是有序的。这意味着两个内容一样的数组如果顺序不同也会被认为是不相等的。在这个例子中，myronsList给"Find a triple rainbow"定的优先级比你高。把这个目标放在myronsList的末尾来让两个清单相等，如代码清单9-16所示。

代码清单9-16　解决myronsList的问题

```
import Cocoa

var bucketList = ["Climb Mt. Everest"]
var newItems = [
                "Fly hot air balloon to Fiji",
                "Watch the Lord of the Rings trilogy in one day",
                "Go on a walkabout",
                "Scuba dive in the Great Blue Hole",
                "Find a triple rainbow"
                ]
bucketList += newItems
bucketList
bucketList.remove(at: 2)
print(bucketList.count)
print(bucketList[0...2])
bucketList[2] += " in Australia"
bucketList[0] = "Climb Mt. Kilimanjaro"
bucketList.insert("Toboggan across Alaska", at: 2)
bucketList

var myronsList = [
                  "Climb Mt. Kilimanjaro",
                  "Fly hot air balloon to Fiji",
                  "Toboggan across Alaska",
                  "Go on a walkabout in Australia",
                  "Find a triple rainbow",
                  "Scuba dive in the Great Blue Hole",
                  "Find a triple rainbow"
                  ]

let equal = (bucketList == myronsList)
```

9.4　不可变数组

你花了很多精力来对目标清单数组进行小修小补，但是也可以创建一个不能修改的数组，也就是不可变数组（immutable array）。下面介绍其使用方法。

假设我们在写一个应用，让用户记录每周吃了什么午饭。用户会记录他们吃了什么以及其他信息，稍后生成一个报告。我们决定把这些用餐记录放在不可变数组中来生成报告。毕竟上周的午饭都已经吃过了，不可能再去修改。

创建一个不可变数组并用一周的午饭初始化，如代码清单9-17所示。

代码清单9-17　不可变数组

```
...
let lunches = [
               "Cheeseburger",
```

```
    "Veggie Pizza",
    "Chicken Caesar Salad",
    "Black Bean Burrito",
    "Falafel Wrap"
]
```

使用`let`关键字创建不可变数组。如果试图以任何方式修改数组，编译器会报错，提示你不能修改不可变数组。如果试图重新赋一个新数组给`lunches`，编译器会报错，提示无法给一个用`let`关键字创建的常量重新赋值。

9.5 文档

任何编程语言的文档都是不可或缺的资源，Swift也不例外。

点击顶部的Help → Documentation and API Reference打开Xcode自带的文档，如图9-3所示。

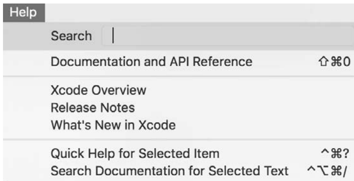


图9-3 帮助菜单

这会打开一个新窗口。在顶部搜索栏输入Array并按回车键，会打开Swift中`Array`类型的文档，如图9-4所示。

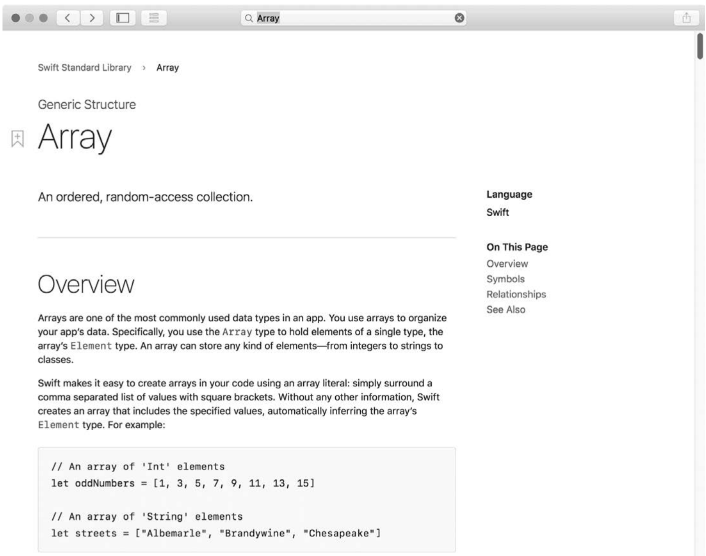


图9-4 打开文档

花些时间探索Array的文档。知道文档的组织方式可以在未来节省很多时间。你会经常访问这些页面的。

9.6 青铜挑战练习

观察下面的数组。

```
var toDoList = ["Take out garbage", "Pay bills", "Cross off finished items"]
```

有一个Array类型的变量会告诉你toDoList是否有元素，利用文档找到它。

9.7 白银挑战练习

把青铜挑战练习中的数组输入playground。用循环反转这个数组的元素顺序，然后把结果输出到控制台。最后研究一下文档，看是否还有更方便的方法完成这个操作。

9.8 黄金挑战练习

我们经常会用到索引来操作数组。查看文档并寻找数组的一个方法，用来定位bucketList中"Fly hot air balloon to Fiji"的索引。

注意：这个方法会返回Index?（也就是可空索引）。先展开这个值，然后用它计算数组中两个位置后的索引。最后，用这个新索引找到bucketList中那个位置上的字符串。

第 10 章

字　　典

我们在上一章熟悉了Swift的Array类型。当容器中的元素顺序很重要时，Array类型很有用。

然而顺序不总是很重要。有时候我们只是想在容器中持有一组数据，并在需要时获取信息。这就是字典（dictionary）的使用场景。

Dictionary使用键值对（key-value pair）组织其内容的容器类型。字典的键映射到值。键就像在衣帽间递给服务员的票，把票给服务员，他就会找到你的大衣。与之类似，把键给Dictionary类型的实例，它就会返回那个键关联的值。

Dictionary中的键必须是唯一的。这个要求意味着每个键都唯一地映射到对应的值。还是用衣帽间打比方，里面可能有好几件海军蓝大衣。只要每件大衣有自己的票，就能确保服务员拿到票后一定能找到你的海军蓝大衣。

本章会介绍如何做到以下几件事：

- 创建并初始化字典
- 遍历字典
- 用键访问并修改字典

我们还会介绍键及其工作原理，尤其是跟Swift相关的部分。最后介绍如何用字典的键值创建数组。

10.1　创建字典

创建Swift字典的通用语法如下：var dict: Dictionary<Key, Value>。这行代码创建一个Dictionary的可变实例，名叫dict。对字典的键和值接受什么类型的声明位于尖括号（<>）中，这里用Key和Value表示。

对于Swift中Dictionary类型的键，唯一的要求是其必须可散列（hashable）：也就是每个Key必须提供一种机制让Dictionary保证任何给定的键都是唯一的。Swift的基本类型都是可散列的，比如String、Int、Float、Double和Bool。

开始写代码之前，先来看看获得一个Dictionary实例的不同方法。

```
var dict1: Dictionary<String, Double> = [:]
var dict2 = Dictionary<String, Double>()
```

```
var dict3: [String:Double] = [:]
var dict4 = [String:Double]()
```

这4种方法得到的是同样的结果：经过完整初始化的Dictionary类型的实例，以及其键值的类型信息。Key被设置为接受String类型的键，而Value被设置为接受Double类型的值。在这4种情况下，字典实例都是空的：没有键也没有值。

[:]和()语法有什么区别呢？本质上是一样的。两者都创建并准备好了Dictionary类型的实例。[:]使用字面量语法创建Dictionary类型的空实例，并且会使用声明中提供的类型信息约束键和值。比如说，dict1指定其类型并被初始化为空字典。()语法则使用Dictionary类型的默认初始化方法，这个方法会准备一个空的字典实例。本书后面部分会详细介绍初始化方法。

利用好Swift的类型推断能力很有用。类型推断让代码更简洁，而且表达力不变。因此，本章会坚持使用类型推断。

10.2　填充字典

开始前，先创建一个新的playground，命名为Dictionary。声明一个名为movieRatings的字典，利用类型推断用一些数据将其初始化，如代码清单10-1所示。

代码清单10-1　创建字典

```
import Cocoa

var str = "Hello, playground"

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
```

我们用Dictionary的字面量语法创建了可变字典，这个字典会持有电影评分数据。字典的键是String，表示一部电影。这些键映射到的值是Int，表示一部电影的评分。

10.3　访问和修改字典

现在有了可变字典，该怎么用呢？你需要从字典中读取和修改数据。首先从利用count获取字典的有用信息开始，如代码清单10-2所示。

代码清单10-2　使用count

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
```

可以在控制台看到这句话：I have rated 3 movies。

现在从movieRatings字典中读取值，如代码清单10-3所示。

代码清单10-3　从字典中读取值

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
```

从字典中访问值，需要提供与要获取的值关联的键。上例给电影评分字典提供了"Donnie Darko"这个键。darkoRating现在等于4。

按住Option并点击darkoRating实例来获取更多信息（如图10-1所示）。

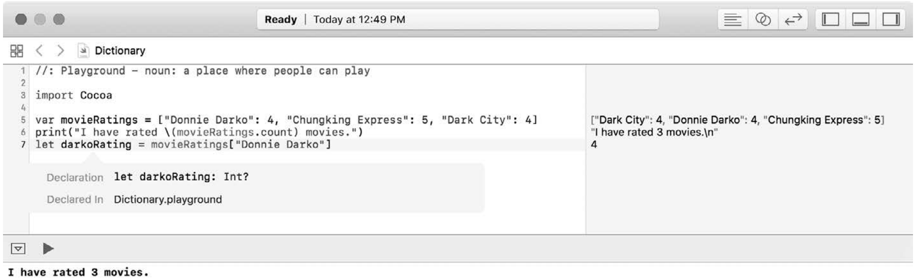


图10-1　按住Option并点击darkoRating

Xcode告诉我们其类型是Int?，但是movieRatings的类型是[String:Int]。为什么会不一样？Dictionary类型需要有一种方法告诉你请求的值不存在。比如说，现在还没有《勇敢的心》（*Braveheart*）的评分，所以代码：let braveheartRating = movieRatings["Braveheart"]会使braveheartRating的类型为Int?并且被置为nil。

上面用方括号包围了下标访问movieRatings：movieRatings["Donnie Darko"]。这种语法向字典请求与String键"Donnie Darko"关联的值。无论何时用下标访问Dictionary实例的给定键，字典都会返回与Dictionary值的类型相匹配的可空类型。本例中，用下标访问movieRatings的给定键会返回Int?（一个可空Int）。

接下来修改电影评分字典中的值，如代码清单10-4所示。

代码清单10-4　修改值

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
```

如你所见，与键"Dark City"关联的值现在等于5。

还有一种有用的方式可以更新与字典的键相关联的值：updateValue(_:forKey:)方法。这个方法接受两个参数：value: Value和forKey: Key。第一个参数value接受新的值，第二个参数forKey指定哪个键需要改变值。

这个方法之所以有用，是因为它能保存该键之前映射的值。还有个小小的警告：updateValue(_:forKey:)返回可空类型。这种返回类型很方便，因为这个键可能在字典中不存在。因此，把updateValue(_:forKey:)的返回值赋给一个预期类型的可空实例，并用可空实例绑定来获取这个键的旧值会很有用。我们来看看实际应用（参见代码清单10-5）。

代码清单10-5　更新值

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
```

图10-2显示了运行结果侧边栏中《死亡幻觉》（*Donnie Darko*）的旧值和新值。



图10-2　更新过的值

10.4　增加和删除值

看过了如何更新值，我们再来看一下如何通过增加和删除值来更新字典中的键值对。首先从增加值开始，如代码清单10-6所示。

代码清单10-6　增加值

```
import Cocoa
```

```
var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
```

这里用语法`movieRatings["The Cabinet of Dr. Caligari"] = 5`给字典增加了新的键值对。赋值运算符将值（本例中是5）关联到了新键（`"The Cabinet of Dr. Caligari"`）上。

接下来删除《移魂都市》（*Dark City*）条目，如代码清单10-7所示。

代码清单10-7 删除值

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings.removeValue(forKey: "Dark City")
```

`removeValue(forKey:)`方法接受一个键作为参数，将与其匹配的键值对删除。现在`movieRatings`已经没有《移魂都市》条目了。

此外，如果键存在并且已成功删除，这个方法还会返回与其关联的值。

在上例中，也可以这样写：`let removedRating: Int? = movieRatings.removeValue(forKey: "Dark City")`。因为`removeValue(forKey:)`会返回被删除的实例的可空类型，所以`removeRating`是可空`Int`。如果需要对旧值进行一些处理的话，像这样把旧值放在变量或常量里就很方便。

不过，不一定要把这个方法的返回值赋给任何东西。如果这个键在字典中存在，那么无论是否将旧值赋给任何东西，这个键值对都会被删除。

也可以通过把键的值设为`nil`来删除键值对，如代码清单10-8所示。

代码清单10-8 把一个键的值设为nil

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
```

```
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings.removeValue(forKey: "Dark City")
movieRatings["Dark City"] = nil
```

结果在本质上是一样的，但是这样写不会返回被删除键的值。

10.5　循环

用for-in循环可以遍历字典。Swift的Dictionary类型为遍历实例中每个元素的键值对提供了一种方便的机制。这种机制通过表示键和值的临时常量把每个元素分为其组成部分。这些常量放在一个元组中，for-in循环可以在循环体内访问，如代码清单10-9所示。

代码清单10-9　遍历字典

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
     print("The movie \(key) was rated \(value).")
}
```

注意字符串插值如何把key和value的值组合为一个字符串。你会看到每部电影及其评分都输出到控制台了。

你并不需要同时访问每个元素的键和值。如果只需要其中一个，Dictionary提供了可以单独访问键或值的属性，如代码清单10-10所示。

代码清单10-10　只访问键，谢谢

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
```

```
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
     print("The movie \(key) was rated \(value).")
}
for movie in movieRatings.keys {
     print("User has rated \(movie).")
}
```

新的循环遍历movieRatings的键，并把用户已经评分的每部电影打印到控制台。

10.6　不可变字典

创建不可变字典和创建不可变数组差不多，用let关键字告诉Swift编译器这个Dictionary实例不可变。创建一个不可变字典用来保存一个虚构专辑中的曲目名字及其时长（单位是秒），如代码清单10-11所示。

代码清单10-11　创建不可变字典

```
...
let album = ["Diet Roast Beef": 268,
             "Dubba Dubbs Stubs His Toe": 467,
             "Smokey's Carpet Cleaning Service": 187,
             "Track 4": 221]
```

曲目名字是键，曲目时长是值。如果要改变字典，编译器会报错并阻止改动。（试试看！）

10.7　把字典转换为数组

有时候把字典中的信息取出并放入数组会很有用。举个例子，假设你要把所有评过分的电影列出来（不带评分）。

在本例中，合理的做法是用字典的键创建一个Array，如代码清单10-12所示。

代码清单10-12　把键输入数组

```
import Cocoa

var movieRatings = ["Donnie Darko": 4, "Chungking Express": 5, "Dark City": 4]
print("I have rated \(movieRatings.count) movies.")
let darkoRating = movieRatings["Donnie Darko"]
movieRatings["Dark City"] = 5
movieRatings
let oldRating: Int? = movieRatings.updateValue(5, forKey: "Donnie Darko")
if let lastRating = oldRating, let currentRating = movieRatings["Donnie Darko"] {
```

```
    print("Old rating: \(lastRating); current rating: \(currentRating)")
}
movieRatings["The Cabinet of Dr. Caligari"] = 5
movieRatings["Dark City"] = nil
for (key, value) in movieRatings {
    print("The movie \(key) was rated \(value).")
}
for movie in movieRatings.keys {
    print("User has rated \(movie).")
}
let watchedMovies = Array(movieRatings.keys)
...
```

这段代码用`Array()`语法创建了一个新的`[String]`实例。圆括号内传递的是字典的键。结果是`watchedMovies`是`Array`常量，表示已经存在于`movieRatings`字典中的用户的电影。

10.8　白银挑战练习

把`Array`放进字典并不罕见。创建一个表示美国某州的字典，键表示郡（为保持简单，只放三个郡）。每个键会映射到一个数组，数组持有该郡的5个邮编。（可以编造郡名和邮编。）

最后，只输出字典的邮编。结果看起来应该类似下面这样。注意，我们把邮编的格式调整了一下，以免超出书页的范围。你还是可以在一行里打印邮编。

```
Georgia has the following zip codes: [30306, 30307, 30308, 30309, 30310,
                                      30311, 30312, 30313, 30314, 30315,
                                      30301, 30302, 30303, 30304, 30305]
```

10.9　黄金挑战练习

再做一遍白银挑战练习，不过这次打印出来的邮编格式要跟上面的示例一样。你可能需要在文档中查找如何用字符串字面量表示特殊字符。（举个例子，如何在字符串中表示换行符？）你可能还需要查看`print()`函数的文档来使用参数`terminator`，这个参数默认在控制台打印的字符串后面添加换行符，但是这里需要别的行为。

完成这个练习的方法有很多，但是不过其中一种方法是打印某些邮编时打印一个带一定数量空格的字符串。（具体多少空格需要你自己计算！）

第 11 章 集　　合

Swift提供的第三种容器类型是集合（set）。Set不是很常用，纯粹只是因为约定俗成而提供的，但是我们认为实际情况不是这样。本章会介绍Set并展示其独特的优势。

11.1　什么是集合

集合是一组互不相同的实例的无序组合。这个定义将其与数组区别开来，后者是有序的，并且可以容纳重复的值。比如说，数组可以有类似[2,2,2,2]的内容，但是集合不行。

集合与字典有很多相似之处，但是又有一些区别。跟字典一样，集合内的值是无序的。字典的键必须唯一，集合也不允许有重复值。为了确保元素唯一，集合需要其元素符合Hashable协议，就跟字典的键一样。不过，字典的值可以通过对应的键来访问，而集合只是存储了单个元素而不是键值对。

表11-1总结了这些相同点和不同点。

表11-1　比较Swift的容器

容器类型	有序	唯一	存储
数组	是	否	元素
字典	否	键	键值对
集合	否	元素	元素

11.2　创建集合

现在创建一个Set实例。创建一个新的playground，命名为Groceries。

输入如代码清单11-1所示代码来生成一个Set实例。

代码清单11-1　创建集合

```
import Cocoa

var str = "Hello, playground"

var groceryBag = Set<String>()
```

我们创建了一个Set实例，声明其持有String。这是一个名为groceryBag的可变集合，目

前是空的。我们来添加点东西。

可以用insert(_:)方法给groceryBag加入食物，如代码清单11-2所示。

代码清单11-2　给集合增加元素

```
...
var groceryBag = Set<String>()
groceryBag.insert("Apples")
groceryBag.insert("Oranges")
groceryBag.insert("Pineapple")
```

现在groceryBag里面有了些东西。跟字典和数组一样，我们可以遍历集合查看其内容，如代码清单11-3所示。

代码清单11-3　遍历集合

```
...
var groceryBag = Set<String>()
groceryBag.insert("Apples")
groceryBag.insert("Oranges")
groceryBag.insert("Pineapple")

for food in groceryBag {
    print(food)
}
```

如果打开调试区域，你会看到groceryBag中的每个元素都打印到了控制台。

截至Swift 3.0，Set还没有自己的字面量语法。不过还是可以用比上面更方便的语法创建集合。假设我们已经知道了在创建Set实例时要添加进去的实例，如代码清单11-4所示。

代码清单11-4　再次创建集合

```
...
var groceryBag = Set<String>(["Apples", "Oranges", "Pineapple"])
groceryBag.insert("Apples")
groceryBag.insert("Oranges")
groceryBag.insert("Pineapple")

for food in groceryBag {
    print(food)
}
```

这段代码使用集合的初始化方法从一个Array实例创建一个Set实例（第17章会深入讲解初始化方法）。这样，就不需要调用三次insert(_:)方法。

集合提供了另一种更方便的语法创建实例。这种语法把实例为Set类型的声明和Array的字面量语法结合起来。比如说，代码清单11-4中的代码可以用下面的例子替换。

```
...
var groceryBag = Set(["Apples", "Oranges", "Pineapple"])
var groceryBag: Set = ["Apples", "Oranges", "Pineapple"]
```

```
for food in groceryBag {
    print(food)
}
```

这段代码显式声明groceryBag是Set类型。这意味着我们可以用Array字面量语法创建一个Set实例。

11.3　运用集合

现在有了Set实例,你可能会想知道如何处理其内部的元素。比如,你可能想知道groceryBag是否包含某个特定元素。Set类型提供了contains(_:)方法来查看其内部是否有某个特殊的元素，如代码清单11-5所示。

代码清单11-5　有香蕉吗

```
...
var groceryBag: Set = ["Apples", "Oranges", "Pineapple"]

for food in groceryBag {
    print(food)
}

let hasBananas = groceryBag.contains("Bananas")
```

hasBananas的值为假，你的groceryBag里面没有香蕉。

11.3.1　并集

想象一下你在食品店闲逛，正好遇到一个朋友。你们聊了起来，朋友提议一起买东西。你瞄了一眼她的购物车，发现她拿了香蕉。因为你正在找香蕉以完成自己著名的水果沙拉配方，所以决定把你们的食品袋合并起来，如代码清单11-6所示。

代码清单11-6　合并集合

```
...
var groceryBag: Set = ["Apples", "Oranges", "Pineapple"]

for food in groceryBag {
    print(food)
}

let hasBananas = groceryBag.contains("Bananas")
let friendsGroceryBag = Set(["Bananas", "Cereal", "Milk", "Oranges"])
let commonGroceryBag = groceryBag.union(friendsGroceryBag)
```

这段代码增加了一个新的Set常量来表示你朋友的食品袋，并且用union(_:)方法把两个集合合并起来。union(_:)是Set类型的一个方法，接受一个SequenceType参数，并返回一个新的Set实例。该实例包含了两个容器中去重后的元素。简单地说，你可以把字典和集合传给union(_:)

并得到一个包含合并后元素的集合，不会有重复。这里，commonGroceryBag就是那个包含groceryBag和friendsGroceryBag去重后元素的集合。

图11-1用图形化的方式说明了两个集合的并集。

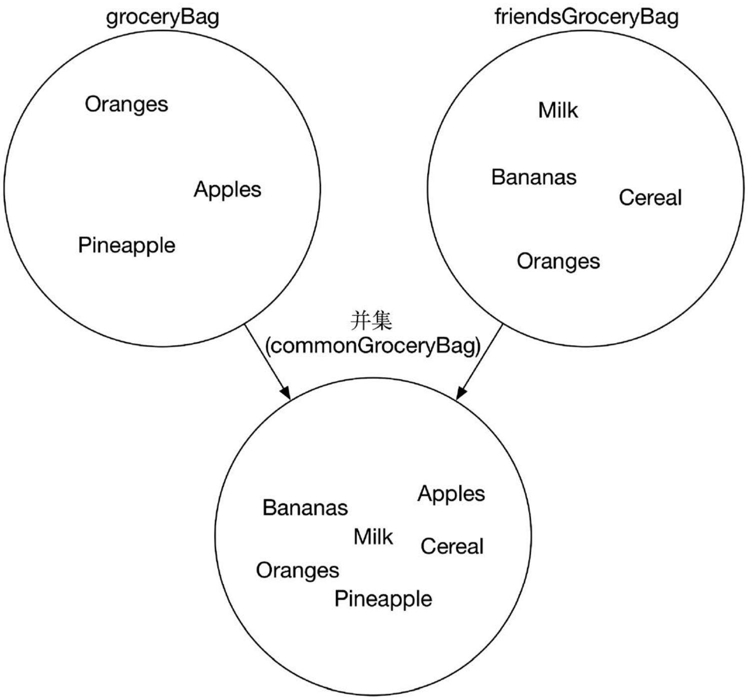


图11-1　两个集合的并集

11.3.2　交集

你和朋友买完食品后去你家准备做那道著名的水果沙拉。到家后，你发现室友也刚从食品店回来，而他刚好也想做水果沙拉。因此，你们比较了一下彼此的食品袋来弄清楚哪些食品是重复的，以便退回食品店，如代码清单11-7所示。

代码清单11-7　集合的交集

```
...
var groceryBag: Set = ["Apples", "Oranges", "Pineapple"]

for food in groceryBag {
    print(food)
}

let hasBananas = groceryBag.contains("Bananas")
```

```
let friendsGroceryBag = Set(["Bananas", "Cereal", "Milk", "Oranges"])
let commonGroceryBag = groceryBag.union(friendsGroceryBag)

let roommatesGroceryBag = Set(["Apples", "Bananas", "Cereal", "Toothpaste"])
let itemsToReturn = commonGroceryBag.intersection(roommatesGroceryBag)
```

集合提供了intersection(_:)方法来找出同时存在于两个容器中的元素，并用一个新的Set实例返回这些重复的元素。图11-2用韦恩图的形式说明了这种关系。你室友的食品袋和你的有几样重复，不过不是所有都重复。

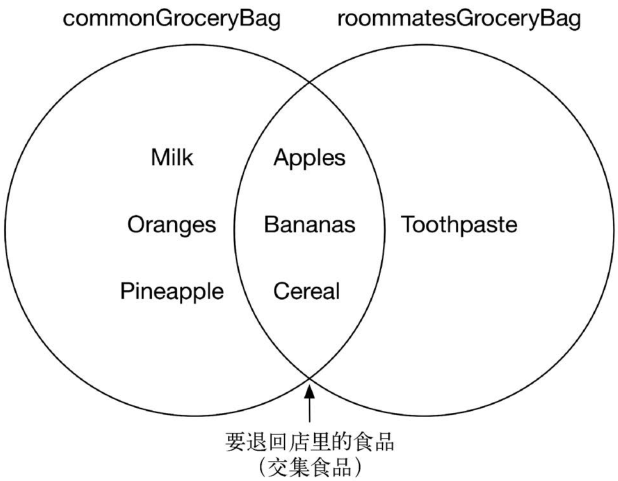


图11-2　集合的交集

11.3.3　不相交

我们讲到了如何用union(_:)方法把两个集合合并为一个包含所有元素的新集合，也讲到了如何用intersection(_:)方法找出两个集合的共同元素，并将它们放进新的集合。如果你想知道两个集合是否包含共同元素，又该怎么办呢？

举个例子，考虑这种情景：你和室友发现你们都忘了著名水果沙拉所需的一些原料。你让朋友开始切水果，你和室友回到店里去买最后的几样原料（还要把重复的退掉）。你们计划在店里分头行动，找到不同的食品，以便尽快跑完这一趟。如果你们能在收银台碰头并快速比较一下购物车以确保没买重复，那不是很棒吗？Swift的Set类型有一个方便的方法来帮助你做到这一点，如代码清单11-8所示。

代码清单11-8　检测集合的交集

```
...
var groceryBag: Set = ["Apples", "Oranges", "Pineapple"]
```

11

```
for food in groceryBag {
    print(food)
}

let hasBananas = groceryBag.contains("Bananas")
let friendsGroceryBag = Set(["Bananas", "Cereal", "Milk", "Oranges"])
let commonGroceryBag = groceryBag.union(friendsGroceryBag)

let roommatesGroceryBag = Set(["Apples", "Bananas", "Cereal", "Toothpaste"])
let itemsToReturn = commonGroceryBag.intersect(roommatesGroceryBag)

let yourSecondBag = Set(["Berries", "Yogurt"])
let roommatesSecondBag = Set(["Grapes", "Honey"])
let disjoint = yourSecondBag.isDisjoint(with: roommatesSecondBag)
```

你决定买浆果和酸奶，你的室友买葡萄和蜂蜜。你们俩在收银台集合并检查购物车确保没有买重复。根据是否有集合（如yourSecondBag）的任何成员出现在作为参数提供给isDisjoint的序列（如roommatesSecondBag）中，Set的isDisjoint(with:)方法会返回真或假。

在本例中，disjoint为真。两个集合（食品袋）没有任何食品重复。你和室友可以结账然后回家做水果沙拉了。

11.4　青铜挑战练习

观察如下代码，这段代码用集合模拟两个人去过的城市。

```
let myCities = Set(["Atlanta", "Chicago",
                    "Jacksonville", "New York", "San Francisco"])
let yourCities = Set(["Chicago", "San Francisco", "Jacksonville"])
```

找到Set的一个方法，根据myCities是否包含所有yourCities中的城市来返回布尔值。[提示：这种关系使得myCities成为yourCities的超集（superset）。]

11.5　白银挑战练习

本章用到了union(_:)和intersection(_:)等方法来创建新集合。不过有时候，你可能并不想创建新实例，而是想在已有的实例上原地改动。查看文档，找到Set类型的合适方法。重新修改本章的示例代码，把union(_:)和intersection(_:)替换为这些方法。

第 12 章

函　数

函数（function）是一组有名字的代码，用来完成某个特定的任务。函数的名字描述了其执行的任务。前面已经用过一些函数，比如Swift提供的`print()`，以及我们所写代码创建的其他函数。

函数会执行代码。有些函数会定义参数，用来传递数据以帮助函数完成工作。有些函数在完成工作后会返回一些信息。可以把函数理解为一部小机器，打开后，它就开始运转并完成自己的工作。如果它的工作方式需要数据的话，就要给它传入数据，然后它会返回一块新数据作为工作成果。

函数是编程中非常重要的一部分。实际上，程序在很大程度上就是一组相关的函数共同完成某种功能。因此，本章的内容很多。慢慢来，调整好心态，对新概念自信应对，然后再继续。

我们从一些例子开始。

12.1　一个基本的函数

创建一个名为Functions的playground。输入如代码清单12-1所示代码。

代码清单12-1　定义函数

```
import Cocoa

var str = "Hello, playground"

func printGreeting() {
     print("Hello, playground.")
}
printGreeting()
```

这段代码用`func`关键字后跟函数名字`printGreeting()`来定义一个函数。圆括号是空的，因为这个函数不接受任何参数。（稍后会详细介绍参数。）

左花括号（`{`）代表函数实现的开始。你可以在这里写代码，描述函数如何工作。调用函数时，花括号中的代码会执行。`printGreeting()`函数非常简单，只有一行代码，利用`print()`来打印`Hello, playground.`到控制台。

最后调用函数，让它执行内部的代码。要做到这一点，在函数定义下一行输入函数名

printGreeting()。调用函数会执行其代码，然后Hello, playground.就被输出到控制台了。

现在你已经写下并执行了一个简单的函数，可以升级到更复杂一点的函数了。

12.2　函数参数

函数有参数（parameter）之后就能做更多的事情了。利用参数可以向函数输入数据。我们之所以把函数的这部分称为“参数”，是因为它们可以根据调用者给函数传递的数据来改变自己的值。函数利用传递给自己的参数来执行任务或产生结果。

创建一个函数，利用参数打印更加个性化的问候信息，如代码清单12-2所示。

代码清单12-2　使用参数

```
import Cocoa

func printGreeting() {
    print("Hello, playground.")
}
printGreeting()

func printPersonalGreeting(name: String) {
    print("Hello \(name), welcome to your playground.")
}
printPersonalGreeting(name: "Matt")
```

printPersonalGreeting(name:)接受一个参数，如函数名后面的圆括号中所示。实参（argument）是调用者传递给函数形参（parameter）的值。本例的函数有一个String类型的形参name。在紧随着name的:后面指定其类型，跟指定变量和常量的类型一样。

快速补充一下，术语形参和实参从技术角度讲是不同的，不过有些人会不加区分地使用。还有，你可能想知道为什么printPersonalGreeting(name:)的括号里要写name:。这表示printPersonalGreeting(name:)有一个形参，在调用的时候要用到其名字，就像printPersonalGreeting(name: "Matt")这样。在调用这个函数的时候必须使用name。下面会详细介绍参数的可见性。

如果传递给形参name的实参是String，那么这个字符串会被插入打印到控制台的字符串。去查看一下，控制台应该显示类似于Hello Matt, welcome to your playground.的信息。

如果不小心传递了非String类型的实参，编译器会报错，告诉你传递的参数错误。这个功能很有用，能让你在写函数实现时知道输入应该是什么样的。

函数可以接受多个参数，而且这种情况常常发生。创建一个做数学运算的新函数，如代码清单12-3所示。

代码清单12-3　做除法的函数

```
...
func printPersonalGreeting(name: String) {
    print("Hello \(name), welcome to your playground.")
```

```
}
printPersonalGreeting(name: "Matt")

func divisionDescriptionFor(numerator: Double, denominator: Double) {
    print("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
```

函数divisionDescriptionFor(numerator:denominator:)描述了对Double类型实例所做的除法。Double是通过两个参数（numerator和denominator）传递的。注意，我们在打印到控制台的字符串的\()中进行了一些数学运算。控制台应该会打印9.0 divided by 3.0 equals 3.0。

12.2.1 参数名字

函数的参数有名字。比如，函数divisionDescriptionFor(numerator:denominator:)有两个参数，参数名分别是numerator和denominator。在用divisionDescriptionFor(numerator:denominator:)时，我们同时用到了两个参数的名字。这是因为在默认情况下调用函数时会用到所有的参数名。

有时候让函数体外可见的参数名不同于内部也是有用的。也就是说调用函数的时候用一个参数名字，在函数体内用另一个名字。这种参数被称为外部参数。

外部参数能让函数可读性更高——如果名字起得合适的话。眼下在调用printPersonalGreeting(name:)时可见的参数名还算能提供有用的信息，但是可读性并不强。通常应该让代码读起来跟我们平常说话一样。

这样写代码就很容易读。举个例子，如果要将函数用在应用代码库的其他文件中，而且函数的实现不是马上可见的、也无法直接猜到，那么推断给函数参数传递什么值就会变得困难。 这会让函数不那么好用，所以在函数里使用更加具有描述性的外部参数名是有用的。

如代码清单12-4所示，更新printPersonalGreeting(name:)，为它添加一个不同于函数内部参数名的外部参数名，让调用这个函数的可读性更强。

代码清单12-4 使用显式的参数名

```
...
func printPersonalGreeting(name: String) {
     print("Hello \(name), welcome to your playground.")
}
printPersonalGreeting(name: "Matt")

func printPersonalGreeting(to name: String) {
     print("Hello \(name), welcome to your playground.")
}
printPersonalGreeting(to: "Matt")

func divisionDescriptionFor(numerator: Double, denominator: Double) {
    print("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
```

现在printPersonalGreeting(to:)有了一个外部参数to:。这个参数能让函数读起来更像日常说话："Print personal greeting to Matt."在函数内，还是得使用name。这样实际上很好，在函数内部，name的含义很清楚。如果在printPersonalGreeting的内部出现print("Hello \(to), welcome to your playground.")的话反而会让人困惑。

你可能注意到了divisionDescriptionFor的末尾是介词。为什么这个函数的末尾有介词，而printPersonalGreeting(to:)的介词在括号内？答案是，Swift的命名指南建议如果一个函数有多个参数形成单一的概念，那么介词应该放在函数名的末尾。这就是divisionDescriptionFor(numerator:denominator)的情形，因为numerator和denominator一起形成分数。另一方面，printPersonalGreeting(to:)没有多个参数，所以介词应该以外部参数名的形式出现在圆括号内。

为函数和参数命名可能会很难，更像是艺术而不是科学。一般来说，我们建议给函数和参数起易读又能提供足够信息的名字。你还应该争取让代码符合日常说话的语法。最后，应该总是考虑函数名字是否容易输入。

12.2.2　变长参数

变长（variadic）参数接受零个或更多输入值作为实参。函数只能有一个变长参数，而且一般应该是参数列表中的最后一个。参数值在函数内部以数组的形式可用。

要声明变长参数，用参数类型后面的三个点表示，如names: String...。在本例中，name在函数体内可用，类型是[String]。

更新printPersonalGreeting(to:)函数来引入变长参数，如代码清单12-5所示。

代码清单12-5　问候一群人

```
...
func printPersonalGreeting(name: String) {
     print("Hello \(name), welcome to your playground.")
}
printPersonalGreeting(to: "Matt")

func printPersonalGreetings(to names: String...) {
    for name in names {
        print("Hello \(name), welcome to the playground.")
    }
}
printPersonalGreetings(to: "Alex","Chris","Drew","Pat")
...
```

现在printPersonalGreeting(to:)函数被一个复数形式的版本代替了：printPersonalGreetings(to:)。查看控制台，你会看到函数为变长参数中的每个名字打印了问候语，如图12-1所示。

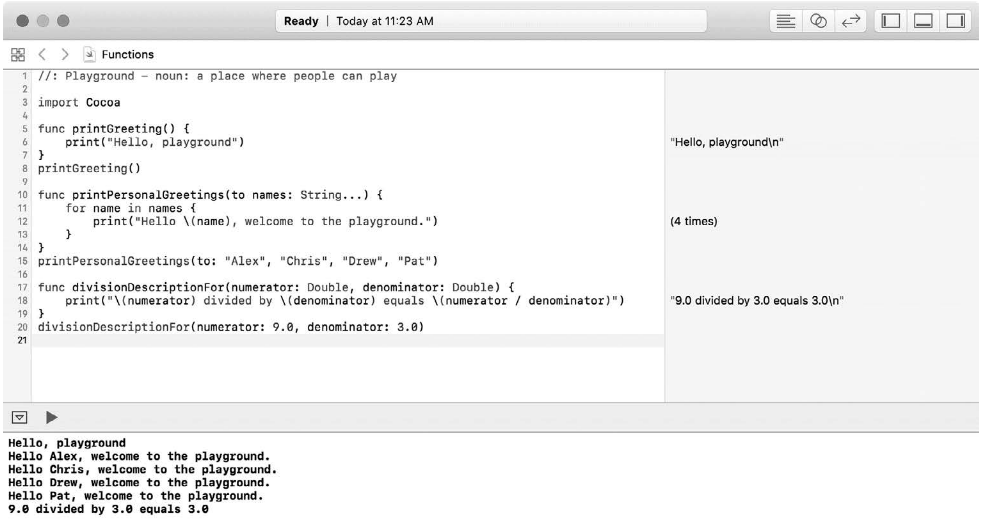


图12-1 多个问候

12.2.3 默认参数值

Swift的参数可以接受默认值。默认值应该放在函数参数列表的末尾。如果形参有默认值，那么在调用函数时可以省略实参。（你可能已经猜到了，函数在这种情况下会使用参数的默认值。）

来看看除法函数中默认参数值的实际应用，如代码清单12-6所示。（注意我们把`print()`的调用折成两行以便能印在书页上。你应该把它输入成一行。）

代码清单12-6 增加默认参数值

```
...
func divisionDescriptionFor(numerator: Double, denonminator: Double) {
    print("\(numerator) divided by \(denominator) equals \(numerator / denominator)")
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
func divisionDescriptionFor(numerator: Double,
                            denominator: Double,
                            withPunctuation punctuation: String = ".") {
    print("\(numerator) divided by \(denominator) equals
           \(numerator / denominator)\(punctuation)")
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
divisionDescriptionFor(numerator: 9.0, denominator: 3.0, withPunctuation: "!")
```

现在函数接受三个参数：`divisionDescriptionFor(numerator:denominator:withPunctuation:)`。注意新增代码：`punctuation: String = "."`。我们为标点增加了一个新参数，加上其期望的类型，并且用`= "."`语法给了它一个默认值。这意味着这个函数创建的字符串默认会以句号结尾。

两次函数调用说明了默认值的用法。要使用默认值，可以像第一次函数调用那样，只省掉最后的参数即可；也可以像第二次函数调用那样，通过传递一个新实参来用新的标点符号替换掉默认值。对函数`divisionDescriptionFor(numerator:denominator:withPunctuation:)`的第一次调用打印了描述信息和句号，第二次则打印了描述信息和感叹号，如图12-2所示。

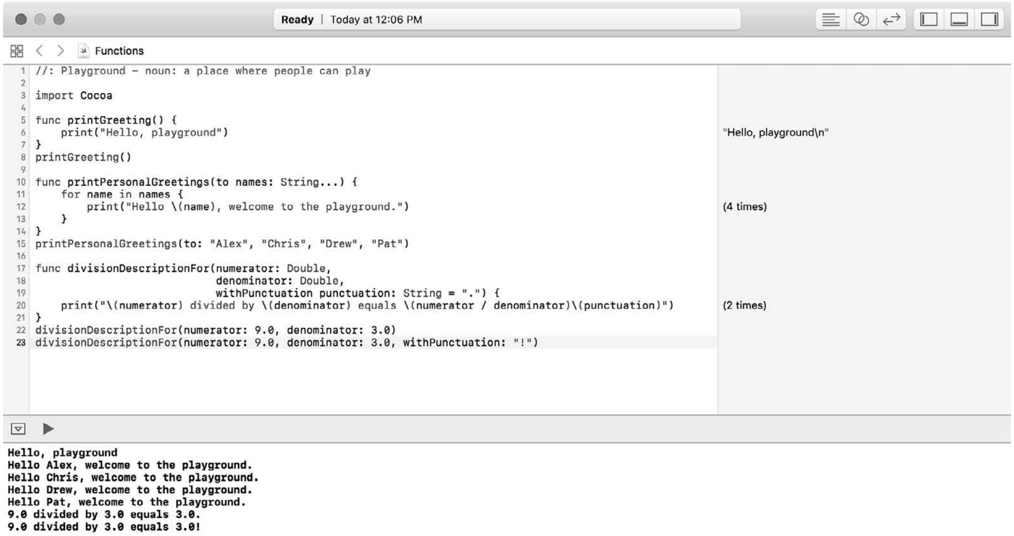


图12-2　默认和指定标点

12.2.4　in-out 参数

出于某种原因，函数有时候需要修改实参的值。in-out参数（in-out parameter）能让函数影响函数体以外的变量。有两个注意事项：首先，in-out参数不能有默认值；其次，变长参数不能标记为`inout`。

假设有一个函数接受一个错误信息作为实参，并根据某些条件在后面添加信息。在playground中输入如代码清单12-7所示的代码。

代码清单12-7　in-out参数

```
...
var error = "The request failed:"
func appendErrorCode(_ code: Int, toErrorString errorString: inout String) {
    if code == 400 {
        errorString += " bad request."
    }
}
appendErrorCode(400, toErrorString: error)
error
```

函数appendErrorCode(_:toErrorString:)有两个参数。第一个是函数要比较的错误码，类型为Int。注意，我们给这个参数的外部名是_。它在Swift中有特殊含义：在参数名前用_会使得函数被调用时省去外部名。由于这个参数名是跟在函数名后面的，在调用的时候没有理由用到这个名字[①]。第二个是命名为toErrorString的inout参数（在名字前用inout关键字标记），类型为String。toErrorString是外部名，用来调用函数；而errorString是内部名，在函数内部使用。

inout加在String前面表示这个函数期望一个特殊的String：它需要一个inout的String。调用这个函数时，传递给inout参数的变量需要在前面加上&。这表示函数会修改这个变量。在这里，errorString被改为The request failed: bad request，可以在运行结果侧边栏看到。

in-out参数和函数返回值不同。如果你的目标是让函数有所产出，那么有更优雅的方式来实现。

12.3 从函数返回

函数结束执行后可以返回一些信息。这些信息称为函数的*返回值*（return）。常见的情况是写下一个函数来完成一些工作，然后返回一些数据。让divisionDescriptionFor(numerator:denominator:withPunctuation:)函数返回一个String类型的实例，如代码清单12-8所示。

代码清单12-8 返回字符串

```
...
func divisionDescriptionFor(numerator: Double,
                            denominator: Double,
                            withPunctuation punctuation: String = ".") {
    print("\(numerator) divided by \(denominator) equals
           \(numerator / denominator)\(punctuation)")
}
divisionDescriptionFor(numerator: 9.0, denominator: 3.0)
divisionDescriptionFor(numerator: 9.0, denominator: 3.0, withPunctuation: "!")
func divisionDescriptionFor(numerator: Double,
                            denominator: Double,
                            withPunctuation punctuation: String = ".") -> String {
    return "\(numerator) divided by \(denominator) equals
            \(numerator / denominator)\(punctuation)"
}
print(divisionDescriptionFor(numerator: 9.0,
      denominator: 3.0,
      withPunctuation: "!"))
...
```

新函数的功能和之前差不多，只有一个变化：新实现有返回值。返回值用-> String语法表示，表明函数会返回指定类型的实例。因为要输出字符串到控制台，所以这个函数返回String。返回字符串的细节在函数体内。

① 在函数名后面跟一个无意义的_作为参数名明显不符合人们的正常认知。——译者注

因为divisionDescriptionFor(numerator:denominator:withPunctuation:)返回String，并且print()的参数类型为String，所以可以在print()的调用中再调用除法函数，把字符串输出到控制台。

12.4　嵌套函数和作用域

Swift的函数定义可以嵌套。嵌套函数在另一个函数定义的内部声明并实现。嵌套函数在包围它的函数以外不可用。当你需要一个函数只在另一个函数内部做一些事情时，这个特性很有用。来看一个例子，如代码清单12-9所示。

代码清单12-9　嵌套函数

```
...
func areaOfTriangleWith(base: Double, height: Double) -> Double {
    let numerator = base * height
    func divide() -> Double {
        return numerator / 2
    }
    return divide()
}
areaOfTriangleWith(base: 3.0, height: 5.0)
```

函数areaOfTriangleWith(base:height:)接受两个参数作为底和高，类型是Double。它还会返回一个Double。在函数内部实现中，我们声明并实现了另一个函数，名为divide()。这个函数没有参数，返回一个Double。函数areaOfTriangleWith(base:height:)调用divide()函数并返回结果。

divide()函数还用到了areaOfTriangleWith(base:height:)中定义的常量numerator。这样为什么能行？

这个常量是在divide()的闭合作用域中定义的。函数中花括号（{}）内部的一切都称为被函数的作用域包围。在本例中，常量numerator和函数divide()都被areaOfTriangle(withBase:andHeight:)的作用域包围。

函数的作用域描述了实例或函数的可见性，这是某种范围。任何定义在函数作用域内部的东西都对函数可见，除此以外的一切都超出了函数的可见范围。numerator对函数divide()可见是因为两者共享同一个闭合作用域。

另一方面，因为divide()函数定义在areaOfTriangleWith(base:height:)函数作用域内部，所以在外面是不可见的。如果试图在包围的函数外部调用divide()函数，编译器会报错。可以试一下看看这个错误。

divide()是个很简单的函数。事实上，areaOfTriangleWith(base:height:)不需要它就可以得到同样的结果：return (base * height) / 2。这里要关注的焦点是作用域的原理。第13章有关于嵌套函数的更复杂的例子；不过先别忙翻页，请继续阅读第12章。

12.5　多个返回值

函数可以返回不止一个值。Swift用元组数据类型来做到这一点，第5章已经介绍过了。回忆一下，元组是相关值的有序列表。为了更好地理解元组的用法，我们要让一个函数接受一个整数数组，然后把这些整数分成奇数和偶数，如代码清单12-10所示。

代码清单12-10　区分奇数和偶数

```
...
func sortedEvenOddNumbers(_ numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}
```

首先声明一个叫作sortedEvenOddNumbers(_:)的函数。这个函数接受一个整数数组作为唯一的参数，并返回一个*命名元组*（named tuple）。元组的组成部分是有名字的，可以从这一点看出这是一个命名元组：evens是整数数组，odds也是整数数组。

接着，在函数的内部实现中初始化evens和odds数组，准备存放相应的整数。然后遍历函数参数numbers里的整数数组，每循环一次，就用%运算符检查number的奇偶性。如果结果是偶数，就添加到evens数组；如果不是偶数，就添加到odds数组。

函数写好了，传递一个整数数组调用一下吧，如代码清单12-11所示。

代码清单12-11　调用sortedEvenOddNumbers(_:)

```
...
func sortedEvenOddNumbers(_ numbers: [Int]) -> (evens: [Int], odds: [Int]) {
    var evens = [Int]()
    var odds = [Int]()
    for number in numbers {
        if number % 2 == 0 {
            evens.append(number)
        } else {
            odds.append(number)
        }
    }
    return (evens, odds)
}

let aBunchOfNumbers = [10,1,4,3,57,43,84,27,156,111]
let theSortedNumbers = sortedEvenOddNumbers(aBunchOfNumbers)
print("The even numbers are: \(theSortedNumbers.evens);
       the odd numbers are: \(theSortedNumbers.odds)")
```

首先创建一个Array类型的实例存放一组整数。接着把数组传递给sortedEvenOddNumbers(_:)函数，并将返回值赋给常量theSortedNumbers。因为返回值指定为(evens: [Int], odds: [Int])，所以编译器推断新创建的常量就是这个类型。最后把结果打印到控制台。

注意，我们用到了字符串插值和元组。如果元组成员有名字，就可以通过名字来访问。比如，theSortedNumbers.evens把evens数组的内容插入字符串并输出到控制台。控制台的输出应该是：The even numbers are: [10, 4, 84, 156]; the odd numbers are: [1, 3, 57, 43, 27, 111].。

12.6　可空返回值类型

有时候，我们想让函数返回可空实例。如果一个函数在某些情况下返回nil，在其他情况下返回一个值的话，Swift提供了可空返回值以供使用。

比如，现在需要一个函数在拿到一个人的全名后取出并返回其中间名。因为不是所有人都有中间名，所以这个函数需要一种机制，能在有中间名的情况下返回中间名，而在其他情况下返回nil。用可空类型就能做到这一点，如代码清单12-12所示。

代码清单12-12　使用可空返回值

```
...
func grabMiddleName(fromFullName name: (String, String?, String)) -> String? {
    return name.1
}

let middleName = grabMiddleName(fromFullName: ("Matt",nil,"Mathias"))
if let theName = middleName {
    print(theName)
}
```

这段代码创建了一个名为grabMiddleName(fromFullName:)的函数。这个函数跟之前的有所不同，它接受一个参数：元组类型(String, String?, String)。元组的三个String实例分别是名字、中间名和姓，而中间名为可空类型。

grabMiddleName(fromFullName:)函数的这个参数叫name，有个外部参数名叫fromFullName。在函数内部实现中可以用要返回名字的索引访问这个参数。元组是零索引的，因此用1来访问实参里的中间名。因为中间名可以是nil，所以函数的返回值可空。

然后调用grabMiddleName(fromFullName:)并传递名字、中间名和姓（可以随意改名字）。因为元组的中间名部分声明为String?,所以可以传递nil。元组的名字和姓部分都不能传递nil。

控制台没有打印东西。因为中间名是nil，可空实例绑定中的布尔值不为真，所以print()不会执行。

试试给中间名一个合法的String实例再看看结果。

12.7　提前退出函数

我们在第3章了解了Swift的条件语句，但是还有一个特性没有介绍：guard语句。跟if/else语句一样，guard语句会根据某个表达式返回的布尔值结果来执行代码；但不同之处是，如果某些条件没有满足，可以用guard语句来提前退出函数，这也是其名称的由来。可以把guard语句想象成一种防止代码在某种不当条件下运行的方式。

接着上面的例子，假设要写一个函数来问候这个人。如果有中间名就用中间名，如果没有中间名就用通用的方式，如代码清单12-13所示。

代码清单12-13　利用guard语句提前退出

```
...
func greetByMiddleName(fromFullName name: (first: String,
                                           middle: String?,
                                           last: String)) {
    guard let middleName = name.middle else {
        print("Hey there!")
        return
    }
    print("Hey \(middleName)")
}
greetByMiddleName(fromFullName: ("Matt","Danger","Mathias"))
```

greetByMiddleName(fromFullName:)类似于grabMiddleName(fromFullName:)，都接受一样的参数；但不同的是前者没有返回值。另一个区别是元组name中的元素有名字，跟人名的各个部分一致。如你所见，这些元素名字在函数内部可用。

guard let middleName = name.middle这行代码把middle的值绑定到middleName常量上。如果可空实例没有值，那么guard语句中的代码会执行。这样会在控制台打印一句省略了中间名的问候语：Hey there!。之后，用return从函数显式返回，这表示guard语句所要求的条件没有满足，函数需要提前返回。

可以把gurad语句想象成防止以下尴尬情况发生：我们在不知道某人中间名的情况下只能含糊应付过去。不过如果元组带着中间名传递给函数，那么其值会绑定到middleName上，并且在guard语句后可用。这意味着middleName在包围着gurad语句的父作用域中可见。

不过，在调用greetByMiddleName(fromFullName:)函数时，会给元组name一个中间名并传递给函数。这意味着控制台会打印“Hey Danger!”。如果中间名是nil，那么控制台会打印"Hey there!"。（自己动手试试吧！）

12.8　函数类型

你在本章中用过的每个函数都有特定的类型。事实上，所有的函数都有。函数类型（function type）由函数参数和返回值组成。以sortedEvenOddNumbers(_:)函数为例，它接受整数数组作为参数，返回有两个整数数组的元组。于是，sortedEvenOddNumbers(_:)的类型可以表示为：

([Int]) -> ([Int], [Int])。

函数参数在左边的圆括号中列出，返回值类型跟在->后面。可以这么读这个函数类型："一个接受整数数组作为参数并返回带有两个整数数组的元组的函数。"作为比较，既没有参数也没有返回值的函数类型是：() -> ()。

函数类型很有用，我们可以把函数类型实例赋给变量。在下一章中，当你看到函数可以作为参数和其他函数的返回值时，会意识到这个特性尤其有用。就目前而言，只要知道如何把函数类型实例赋给常量就可以了。

```
let evenOddFunction: ([Int]) -> ([Int], [Int]) = sortedEvenOddNumbers
```

这行代码创建了一个evenOddFunction常量，其值是sortedEvenOddNumbers(_:)函数。很酷，是吧？现在可以传递常量了，就跟普通常量一样；甚至可以用这个常量来调用函数。举个例子，evenOddFunction(numbers: [1,2,3])会把作为参数传递的数组中的数字放进有两个数组的元组中——一个数组放奇数，另一个放偶数。

你在本章学到了很多。本章有很多内容，我们建议你再读一遍。确保把所有的代码都敲出来。最好试着针对各种情况扩展代码示例，试着把代码改出问题再修复。

如果还是对函数有点迷糊，也别担心。下一章的重点也是函数，还有很多机会练习。

12.9　青铜挑战练习

跟if/else条件语句一样，guard语句支持多个子句做额外的检查。用额外的子句配合guard语句可以对语句的条件有更好的控制。重构greetByMiddleName(name:)函数，在guard语句中增加一个额外的子句。这个子句应该检查中间名是否少于4个字符；如果是，就用中间名问候这个人，否则用通用问候语。

12.10　白银挑战练习

写一个名为siftBeans(fromGroceryList:)的函数，它接受一个食品清单（字符串数组）并把豆子筛选出来。这个函数接受一个名为list的参数，并返回一个类型为(beans: [String], otherGroceries: [String])的命名元组。

下面是一个例子，说明了如何调用函数以及返回值应该是什么。

```
let result = siftBeans(fromGroceryList: ["green beans",
                                         "milk",
                                         "black beans",
                                         "pinto beans",
                                         "apples"])

result.beans == ["green beans", "black beans", "pinto beans"] // 真
result.otherGroceries == ["milk", "apples"] // 真
```

提示：可能需要用到String类型的hasSuffix(_:)函数。

12.11　深入学习：Void

我们在本章中写的第一个函数是printGreeting()。它没有参数也不返回值；还是说，它其实有返回值?

实际上，没有显式返回值的函数还是有返回值，返回的是Void。编译器会帮你在代码中插入这个返回值。

所以当你像下面这样写printGreeting()时：

```
func printGreeting() {
    print("Hello, playground.")
}
```

编译器实际上会在代码中加点东西：

```
func printGreeting() -> Void {
    print("Hello, playground.")
}
```

换句话说，它为你添加了Void返回值。不过Void是什么？像上面这样让printGreeting返回Void。按住Command键并点击Void，Xcode会显示它在标准库中的定义。

```
public typealias Void = ()
```

Void是()的类型别名。本书目前还没有讲到类型别名，不过会在第21章讲到，敬请关注。至于现在，只需要把类型别名理解成告诉编译器某个类型是另一个类型的简略表达的一种方法。在上面的代码片段中，标准库定义了Void是另一种表示()的方式。

这里涉及的概念已经在第5章出现过了。()表示空元组。如果元组是有序元素的列表，那么空元组就是一个空列表。

利用已有的知识，可以知道下面三种printGreeting()的实现是等价的。

```
func printGreeting() {
    print("Hello, playground.")
}

func printGreeting() -> Void {
    print("Hello, playground.")
}

func printGreeting() -> () {
    print("Hello, playground.")
}
```

第一个版本就是原始版本。第二个是编译器自动插入的。第三个用了空的圆括号，它是标准库对Void的映射。

知道Void映射为()能帮你更好地理解某个函数类型。举个例子，printGreeting()的类型是() -> Void。这只是一个没有参数并返回空元组的函数的类型，是所有没有显式返回值的函数的隐式返回类型。

第 13 章

闭　　包

13

闭包（closure）是在应用中完成特定任务的互相分离的功能组。上一章学习的函数是闭包的特殊情况，可以把函数理解为有名字的闭包。

第12章主要用到了全局和嵌套函数。闭包不同于函数的地方在于其语法更加紧凑、轻量。闭包具有“类似于函数”的结构，还能省去命名和函数声明。这让闭包很容易以函数参数和返回值的形式传递。

让我们开始吧。创建一个新的playground，命名为Closures。

13.1　闭包的语法

想象你是一个社区组织者，负责管理若干组织。你想记录每个组织有多少名志愿者，因此创建了一个数组来完成这个任务，如代码清单13-1所示。

代码清单13-1　从数组开始

```
import Cocoa

var str = "Hello, playground"

let volunteerCounts = [1,3,40,32,2,53,77,13]
```

你输入了每个组织提供的志愿者人数，这意味着这个数组是完全无序的。如果志愿者数组能按人数从小到大排序就好了。好消息是，Swift提供了`sorted(by:)`方法来指定如何排列数组。（当一个函数定义在某个类型上时，我们称之为方法，比如这里的`Array`类型。第15章会详细讨论这个话题。）

`sorted(by:)`接受一个参数：一个描述如何排序的闭包。这个闭包接受两个参数，其类型必须和数组元素的类型匹配，并返回布尔值。通过比较两个参数会生成返回值，表示第一个参数是否排在第二个参数前面。如果想让参数一排在参数二前面，可以在返回的时候使用<；这样会把数组按升序（ascending）排列，也就是从小到大排序。如果想让参数二排在参数一前面，可以在返回的时候使用>；这样会把数组按降序（desending）排列，也就是从大到小排序。

因为志愿者人数的数组中都是整数，所以`sorted(by:)`的函数类型应该类似于`((Int, Int) -> Bool) -> [Int]`。读出来就是“`sorted(by:)`是一个接受两个整数进行比较并返回布尔值

表示哪个整数在前的闭包”。sorted(by:)返回一个新的整数数组，这些整数已经根据闭包定义的规则排好序了。

增加如代码清单13-2所示的代码对数组进行排序。

代码清单13-2　对数组排序

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

func sortAscending(_ i: Int, _ j: Int) -> Bool {
    return i < j
}
let volunteersSorted = volunteerCounts.sorted(by: sortAscending)
```

首先，我们创建了函数sortAscending(_:_:)。这个函数比较两个整数并返回一个布尔值表示整数i是否应该在整数j前面。因为sortAscending这个名字已经意味着我们是在对两个实例排序，所以可以用_省去在调用时的参数名。如果i小于j从而应该排在j前面的话，这个函数就会返回true。这个全局函数是一个命名闭包（回忆一下，所有的函数都是闭包），因此可以将其作为sorted(by:)的参数。

接着，调用sorted(by:)，把sortAscending(_:_:)作为第二个参数传递。因为sorted(by:)返回一个新数组，所以需要把结果赋给新的常量数组volunteersSorted。这个实例保存组织内排过序的志愿者人数。

查看playground中的运行结果侧边栏，会看到volunteersSorted中的值是按从小到大排序的（如图13-1所示）。

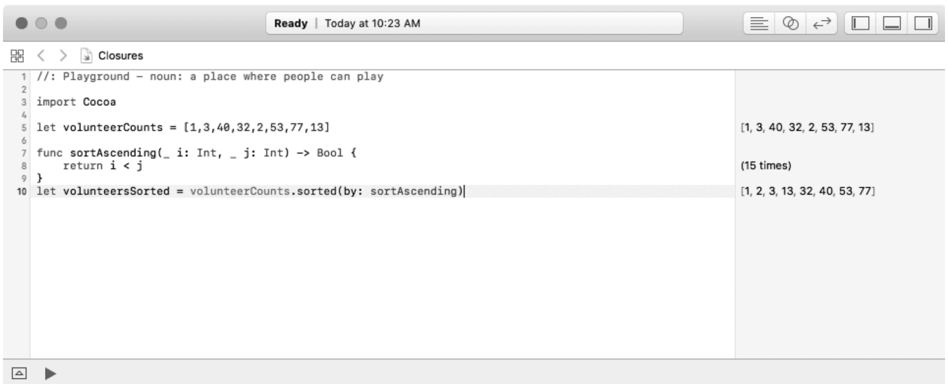


图13-1　对志愿者人数排序

13.2　闭包表达式语法

这样写没问题，但是代码还可以更干净。闭包表达式语法的一般形式如下：

```
{(parameters) -> return type in
    // 代码
}
```

闭包表达式写在花括号（{}）里。紧跟着左花括号的圆括号里是闭包的参数。闭包的返回值类型在参数后面，和常规语法一样。关键字in用来分隔闭包的参数、返回值与闭包体内的语句。

用闭包表达式重构前面的代码：创建一个内联闭包取代在sorted(by:)方法外单独定义的函数，如代码清单13-3所示。

代码清单13-3　重构排序代码

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

func sortAscending(_ i: Int, _ j: Int) -> Bool {
    return i < j
}
let volunteersSorted = volunteerCounts.sorted(by: sortAscending)

let volunteersSorted = volunteerCounts.sorted(by: {
    (i: Int, j: Int) -> Bool in
    return i < j
})
```

这段代码比第一个版本稍微清晰和优雅一些。我们没有提供在playground中其他地方定义的函数，而是在sorted(by:)方法的第二个参数中实现了一个内联闭包。我们在闭包的圆括号中定义了其参数及类型（Int），也指定了返回值类型。接着，用逻辑测试（i是否比j小？）来确定闭包的返回值，从而实现闭包体。

结果跟之前一样：把排好序的数组赋给了volunteersSorted。

这次重构在正确的方向上迈进了一步，但还是有点冗长。闭包可以利用Swift的类型推断系统，因此我们可以通过去除类型信息来进一步简化闭包，如代码清单13-4所示。

代码清单13-4　利用类型推断

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted(by: {
    (i: Int, j: Int) -> Bool in
    return i < j
})

let volunteersSorted = volunteerCounts.sorted(by: { i, j in i < j })
```

这段代码改动了三处。首先，移除了两个参数和返回值的类型信息。返回值类型可以移除是因为编译器知道检查i < j是否成立会返回布尔值true或false。其次，把整个闭包表达式放到了一行。

最后，移除了关键字`return`。不是所有的闭包语句都可以省略`return`关键字，这里可以是因为只有一个表达式（`i < j`）。如果存在更多表达式，那么显式的`return`就是必需的。

注意，侧边栏中的结果没有变。

这个闭包现在变得很紧凑，但是还可以继续优化。Swift提供了快捷参数名，可以在内联闭包表达式中引用。这些快捷参数名和显式声明的参数类似：类型和值都一样。编译器的类型推断能力让它知道闭包接受的参数个数和类型，这意味着不需要给参数命名。

举个例子，编译器知道`sorted(by:)`接受一个闭包。这个闭包本身又接受两个参数，它们的类型和`sorted(by:)`方法的数组参数中的元素类型一样。因为闭包有两个参数，我们能比较其值来判断顺序，所以可以用`$0`引用第一个参数的值，用`$1`引用第二个参数的值。

调整代码来利用快捷语法，如代码清单13-5所示。

代码清单13-5　利用参数的快捷语法

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted(by: { i, j in i < j })

let volunteersSorted = volunteerCounts.sorted(by: { $0 < $1 })
```

现在内联闭包表达式利用了快捷参数语法，就不需要像之前声明`i`和`j`那样显式声明参数了。编译器知道闭包参数的类型是正确的，也知道基于<运算符能推断出什么。

顺便提一句，对于多于两个参数的闭包，可以用`$2`、`$3`等。

别觉得这个闭包已经不能更加简洁了，还有改进空间！如果一个闭包是以一个函数的最后一个参数传递的，那么它就可以在函数的圆括号以外内联。因为`sorted(by:)`只接受一个参数，所以根本不需要圆括号。之所以可以省略闭包的参数名，是因为尾部闭包语法允许这么做。

我们来修改一下，如代码清单13-6所示。

代码清单13-6　作为函数最后一个参数的内联闭包

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted(by: { $0 < $1 })

let volunteersSorted = volunteerCounts.sorted { $0 < $1 }
```


这种*尾部闭包语法*（trailing closure syntax）对于闭包体很长的情况特别有用。在这里，尾部闭包只是让我们少输入了两个圆括号。

不得不说，“简洁是智慧的灵魂”。上面的代码虽然简洁，但是效果和之前那个冗长的版本完全一样。毕竟我们真正关心的其实只有一件事（一个整数是否小于另一个整数？），而这个逻辑表达起来很容易。不过，也别过分利用这些技巧。保持代码的可读性和可维护性永远是最重要的。

13.3　函数作为返回值

有了关于函数和闭包的更多经验，再想想第12章讲到了每个函数都有其参数和返回值类型的函数类型。比如，对于一个接受`String`参数并返回`Int`的函数，其类型是`(String) -> Int`。函数类型常用来判断什么样的闭包能满足给定的参数类型或者需要返回什么样的函数。

函数可以返回其他函数作为返回值。还记得Knowhere镇吗？是时候写个函数来改善镇子了。我们想要修几条路，如代码清单13-7所示。

代码清单13-7　回到Knowhere

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(byAddingLights lights: Int,
                    toExistingLights existingLights: Int) -> Int {
        return lights + existingLights
    }
    return buildRoads
}
```

函数`makeTownGrand()`没有参数，就跟你的祖父一样[①]。不过，它会返回一个函数。这个函数接受两个参数（都是整数），并且返回一个整数。在`makeTownGrand()`函数体内实现要返回的函数。

就实现细节而言，返回的函数是嵌套函数`buildRoads(byAddingLights:toExistingLights:)`。`buildRoads`接受两个`Int`参数并返回`Int`，符合`makeTownGrand()`的返回值的声明。

练习使用新函数修几条路来看看实际效果，如代码清单13-8所示。

代码清单13-8　通往Knowhere的路

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted { $0 < $1 }

func maketowngrand() -> (Int, Int) -> Int {
    func buildRoads(byAddingLights lights: Int,
                    toExistingLights existingLights: Int) -> Int {
        return lights + existingLights
    }
    return buildRoads
}
```

---

① 英文原文是take no arguments，既有没有参数的意思，又表示不接受别人的争辩。——译者注

```
var stoplights = 4
let townPlanByAddingLightsToExistingLights = makeTownGrand()
stoplights = townPlanByAddingLightsToExistingLights(4, stoplights)
print("Knowhere has \(stoplights) stoplights.")
```

首先创建一个变量stoplights。之所以把这个实例声明为变量，是因为接下去要修几条路，会增加镇子的红绿灯数量。接着声明常量townPlanByAddingLightsToExistingLights，引用makeTownGrand()创建的buildRoads函数。利用makeTownGrand()的返回类型中所列出的参数名就可以调用这个函数，传入要增加的红绿灯数量（第一个参数）和当前的红绿灯数量（第二个参数）。这个函数的结果是一个Int实例，再次被赋给stoplights变量。最后，把新值打印到控制台。

查看控制台，输出的信息应该是Knowhere has 8 stoplights。

13.4　函数作为参数

函数可以作为其他函数的参数。比如，我们一开始把sortAscending(_:_:)函数作为参数传递给sorted(by:)。

在现实生活中，只有在预算充足的前提下才会在镇上修路。调整前面的makeTownGrand()函数，使其接受一个预算参数和一个条件参数。预算参数就是镇子的预算，而条件参数则计算预算是否足够修新路，如代码清单13-9所示。

代码清单13-9　增加预算考量

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted { $0 < $1 }

func makeTownGrand() -> (Int, Int) -> Int {
    func buildRoads(byAddingLights lights: Int,
                    toExistingLights existingLights: Int) -> Int {
        return lights + existingLights
    }
    return buildRoads
}
var stoplights = 4
let townPlanByAddingLightsToExistingLights = makeTownGrand()
stoplights = townPlanByAddingLightsToExistingLights(4, stoplights)

func makeTownGrand(withBudget budget: Int,
                   condition: (Int) -> Bool) -> ((Int, Int) -> Int)? {
    if condition(budget) {
        func buildRoads(byAddingLights lights: Int,
                        toExistingLights existingLights: Int) -> Int {
            return lights + existingLights
        }
        return buildRoads
    } else {
```

```
            return nil
        }
    }
    func evaluate(budget: Int) -> Bool {
        return budget > 10_000
    }

    var stoplights = 4

    if let townPlanByAddingLightsToExistingLights = makeTownGrand(withBudget: 1_000,
                                                                  condition: evaluate) {
        stoplights = townPlanByAddingLightsToExistingLights(4, stoplights)
    }
    print("Knowhere has \(stoplights) stoplights.")
```

来看一下这里有哪些修改。

第一个变动是新的makeTownGrand(withBudget:condition:)函数，它接受两个参数。第一个是Int类型，表示镇子的预算。第二个是condition，接受一个函数。这个函数判断镇子的预算是否足够，因此它接受一个整型并返回一个布尔值。如果预算足够，那么这个函数会返回true。如果预算不够，那么这个函数会返回false。

你有没有注意到makeTownGrand(withBudget:condition:)的返回值类型变了？现在的返回值类型是((Int, Int) -> Int)?。makeTownGrand()的上个实现返回一个函数，这个函数接受两个整数参数并返回一个整数。在这个修改后的版本中，makeTownGrand(withBudget:condition:)返回同样的函数，只不过是可空的。

为什么？考虑一下预算需求。

makeTownGrand(withBudget:condition:)的实现会运行通过condition参数传入的函数。如果返回结果为真，镇子有足够的预算，那么会创建buildRoads(byAddingLights:toExistingLights:)函数并将其返回。如果预算不够，就不会创建buildRoads(byAddingLights:toExistingLights:)，并且会返回nil。为了处理好nil返回值的可能性，就需要用到可空类型。

我们还创建了evaluate(budget:)函数，这个函数接受一个整数并返回一个布尔值。它的实现代码会计算这个整数是否大于某个阈值（这里随便设置为10 000）。

最后，我们用可空实例绑定来有条件地设置townPlan。如果给makeTownGrand(withBudget:condition:)提供的预算足够多，就会创建buildRoads(byAddingLights:toExistingLights:)函数并返回，然后将其赋给townPlan。这种情况下，镇子的红绿灯数量会增加4。不过，如果预算不够，那么makeTownGrand(withBudget:condition:)会返回nil。这种情况下，镇子的红绿灯数量不变。

查看控制台。不幸的是，镇子的预算太少了。1000的预算显然不够我们所需的10 000。于是makeTownGrand(withBudget:condition:)会返回nil，而buildRoads(byAddingLights:toExistingLights:)永远不会执行。镇子在有能力修新路之前必须省吃俭用了……

好了，省吃俭用了一段时间，镇子现在有足够的钱修几条路了。更新代码，给出更多的预算来看看效果，如代码清单13-10所示。

代码清单13-10　修更多的路

```
import Cocoa

let volunteerCounts = [1,3,40,32,2,53,77,13]

let volunteersSorted = volunteerCounts.sorted { $0 < $1 }

func makeTownGrand(withBudget budget: Int,
                   condition: (Int) -> Bool) -> ((Int, Int) -> Int)? {
    if condition(budget) {
        func buildRoads(byAddingLights lights: Int,
                        toExistingLights existingLights: Int) -> Int {
            return lights + existingLights
        }
        return buildRoads
    } else {
        return nil
    }
}
func evaluate(budget: Int) -> Bool {
    return budget > 10_000
}

var stoplights = 4

if let townPlanByAddingLightsToExistingLights = makeTownGrand(withBudget: 1_000,
                                                              condition: evaluate) {
    stoplights = townPlanByAddingLightsToExistingLights(4, stoplights)
}
if let newTownPlanByAddingLightsToExistingLights
        = makeTownGrand(withBudget: 10_500, condition: evaluate) {
    stoplights = newTownPlanByAddingLightsToExistingLights(4, stoplights)
}
print("Knowhere has \(stoplights) stoplights.")
```

10 500的预算超出了修路的最低需求。现在应该能在侧边栏和控制台看到镇子有8个红绿灯了！

13.5　闭包能捕获变量

闭包和函数能记录在其闭合作用域中定义的变量所封装的内部信息。来看一个例子，假设Knowhere正在繁荣发展。由于人口增长很难预测，我们可以创建一个函数来根据最近的增长更新镇子的人口数据。镇子的规划者会在每增加500人的时候更新人口普查数据，如代码清单13-11所示。

13

代码清单13-11　记录发展

```
...
print("Knowhere has \(stoplights) stoplights.")

func makePopulationTracker(forInitialPopulation population: Int) -> (Int) -> Int {
```

```
    var totalPopulation = population
    func populationTracker(growth: Int) -> Int {
        totalPopulation += growth
        return totalPopulation
    }
    return populationTracker
}

var currentPopulation = 5_422
let growBy = makePopulationTracker(forInitialPopulation: currentPopulation)
```

函数makePopulationTracker(forInitialPopulation:)创建了函数populationTracker(growth:)。makePopulationTracker(forInitialPopulation:)接受一个整数参数，表示要记录的增长数；并且返回一个接受一个参数的函数，这个函数返回一个整数。返回的整数是持续更新的镇子人口数totalPopulation，它一开始的值是传入makePopulationTracker(forInitialPopulation:)的population。

populationTracker(growth:)函数从闭合作用域中捕获了totalPopulation变量的值。populationTracker(growth:)创建后，totalPopulation变量会按照这个函数的参数所指定的大小增加。

以上代码的结果是，我们新建了一个函数growBy(_:)，它接受一个整数作为唯一的参数并追踪currentPopulation。后面对growBy(_:)的调用会提供一个在初始人口数基础上的增加值。多次调用这个函数来练习和测试，如代码清单13-12所示。

代码清单13-12 人口在增长

```
...
print("Knowhere has \(stoplights) stoplights.")

func makePopulationTracker(forInitialPopulation population: Int) -> (Int) -> Int {
    var totalPopulation = population
    func populationTracker(growth: Int) -> Int {
        totalPopulation += growth
        return totalPopulation
    }
    return populationTracker
}

var currentPopulation = 5_422
let growBy = makePopulationTracker(forInitialPopulation: currentPopulation)
growBy(500)
growBy(500)
growBy(500)
currentPopulation = growBy(500) // currentPopulation现在是7422
```

我们调用了growBy(_:)四次来模拟镇子人口增长2000的情形。注意前三次对growBy(_:)的调用没有把结果赋给任何常量和变量。这样没有问题，因为这个函数在内部记录了持续更新的镇子人口总增长数。要更新镇子的人口数，只需在镇子规划者记录了人口增长数后把这个函数的返回值加到currentPopulation上即可。

13.6 闭包是引用类型

闭包是引用类型（reference type）。这意味着当你把函数赋给常量或变量时，实际上是在让这个常量或变量指向这个函数。我们并没有为这个函数创建新的副本。这个事实的一个重要结果是，如果通过新的常量或变量调用这个函数，那么其作用域捕获的任何信息都会改变。

为了解释这一点，创建一个新常量并将其设置为`growBy(_:)`函数，如代码清单13-13所示。

代码清单13-13 复制增长

```
...
func makePopulationTracker(forInitialPopulation population: Int) -> (Int) -> Int {
    var totalPopulation = population
    func populationTracker(growth: Int) -> Int {
        totalPopulation += growth
        return totalPopulation
    }
    return populationTracker
}

var currentPopulation = 5_422
let growBy = makePopulationTracker(forInitialPopulation: currentPopulation)
growBy(500)
growBy(500)
growBy(500)
currentPopulation = growBy(500) // currentPopulation现在是7422
let anotherGrowBy = growBy
anotherGrowBy(500) // totalPopulation现在是7922
```

`anotherGrowBy`现在和`growBy`指向同一个函数，所以调用`anotherGrowBy(_:)`并传入500作为参数时，变量`totalPopulation`会增加500。但是要记住`currentPopulation`没有变，因为我们没有把`anotherGrowBy(_:)`的返回值加到它上面！换句话说，`anotherGrowBy(_:)`只会直接增加其内部变量`totalPopulation`，而函数`populationTracker(growth:)`的作用域会捕获它。图13-2展示了每次调用`populationTracker(growth:)`时`totalPopulation`的变化曲线。

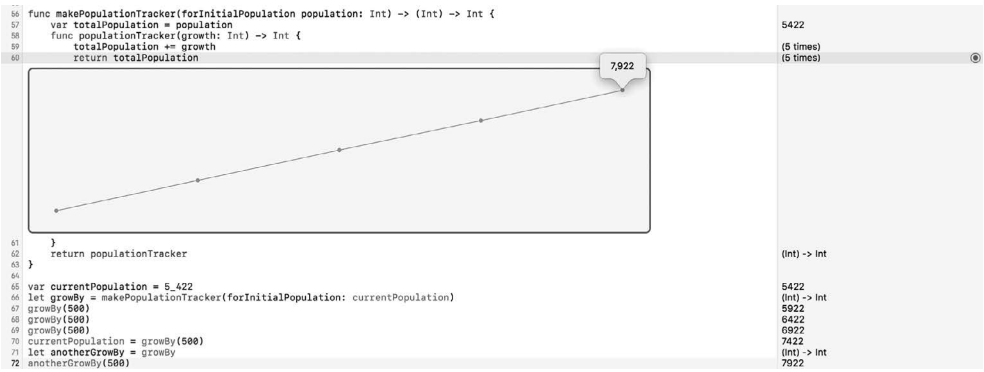


图13-2 totalPopulation的增长

作为对比，假设临近的一座大城市对镇子规划者的函数很感兴趣。这座城市想要有自己的增长记录函数，在每次人口增加10 000时更新人口数据。创建这座城市的人口并用函数makePopulationTracker(forInitialPopulation:)为它创建一个增长记录函数，如代码清单13-14所示。

代码清单13-14　记录另一个城市的人口

```
...
let anotherGrowBy = growBy
anotherGrowBy(500) // totalPopulation现在是7922
var bigCityPopulation = 4_061_981
let bigCityGrowBy = makePopulationTracker(forInitialPopulation: bigCityPopulation)
bigCityPopulation = bigCityGrowBy(10_000)
currentPopulation
```

现在又有了一个要记录的人口数据，并且有了一个新的增长记录函数bigCityGrowBy(_:)来协助记录。用bigCityGrowBy(_:)来增加城市的人口：bigCityPopulation = bigCityGrowBy(10_000)。这跟growBy(_:)的用法类似。城市的人口在这行后增加到4 071 981。

注意currentPopulation 表示的Knowhere镇人口没有变化，还是7422。这是因为我们用makePopulationTracker(forInitialPopulation:)函数创建了一个新的增长记录函数。这个新的增长记录函数是独立的，不同于growBy(_:)。

第18章会详细介绍引用类型（以及值类型；值类型的实例都持有数据的唯一副本）。

13.7　函数式编程

Swift采用了*函数式编程*（functional programming）范式的一些模式。我们很难给函数式编程下一个准确的定义，因为人们用这个词表示不同的含义和意图，但是一般来说这个概念包含以下几点。

- **一等公民函数**：函数可以作为返回值从别的函数返回，也可以作为参数传递给别的函数，可以存储在变量中，等等；就跟其他类型一样。
- **纯函数**：函数没有副作用；给定同样的输入，函数永远返回同样的输出，而且不会修改程序中其他地方的状态。大部分数学函数都是纯函数，像sin、cos、斐波那契和阶乘。
- **不可变性**：不鼓励可变性，因为值可变的数据更难分析。
- **强类型**：强类型系统能增加代码的运行时安全性，因为语言的类型系统的合法性会在编译时得到检查。

Swift支持以上所有特性。

函数式编程能让代码更简洁，更具表达力。通过鼓励不可变性和编译时类型检查，代码在运行时也更安全。函数式编程的这些特点还让代码更易读、更易维护。

Swift的let关键字可以用来声明不可变的实例，其强类型系统能在编译时捕获错误，而不用等到运行时。Swift还提供了一些拥戴函数式编程的开发者所熟知的*高阶函数*（higher-order

function）：map(_:)、filter(_:)和reduce(_:_:)。这些函数进一步说明了Swift函数的确是一等公民。

我们来看看这些函数给Swift工具箱带来了什么。

高阶函数

高阶函数至少接受一个函数作为输入。本章已经用到过高阶函数（比如，前面sorted(by:)的使用）。现在来看看另外三个高阶函数：map(_:)、filter(_:)和reduce(_:_:)。

1. map(_:)

map(_:)可以用来变换数组的内容。你可以把数组的内容从一个值变换成另一个值，并把这些新值放进一个新数组。因为map(_:)是一个高阶函数，所以要给它提供一个函数来告诉它如何变换数组的内容。

Swift的标准库为Array类型提供了map(_:)的实现。假设Knowhere镇有三个选区，每个都有自己的人口。把这些值放进数组precinctPopulations，如代码清单13-15所示。

代码清单13-15 按选区记录人口

```
...
let precinctPopulations = [1244, 2021, 2157]
```

跟之前一样，Knowhere是一个正在发展的镇子。知道Knowhere当前的人口增长速度后，镇子的城市规划者需要预测每个选区的人口。城市规划者可以利用map(_:)配合precinctPopulations数组进行估算，如代码清单13-16所示。

代码清单13-16 用map(_:)估算人口

```
...
let precinctPopulations = [1244, 2021, 2157]
let projectedPopulations = precinctPopulations.map {
    (population: Int) -> Int in
    return population * 2
}
projectedPopulations
```

这段代码用map(_:)对precinctPopulations的每个值进行了估算。（注意尾部闭包语法。）接着声明一个整型参数population，并指定闭包会返回整型。map(_:)会把这个函数应用到precinctPopulations每个索引位置的值上。估算会把每个选区的人口增加200%，结果放在新数组projectedPopulations中，其值是2488、4042和4314。

2. filter(_:)

filter(_:)类似于map(_:)，可以对Array类型进行调用。它也接受一个闭包表达式作为参数，目的是基于某些条件过滤数组。结果数组会包含原数组中通过测试的值。

进行估算后，城市规划者想知道哪个选区的人口多于4000。filter(_:)是完成这个操作的理想选择，如代码清单13-17所示。

代码清单13-17　过滤数组

```
...
let precinctPopulations = [1244, 2021, 2157]
let projectedPopulations = precinctPopulations.map {
    (population: Int) -> Int in
    return population * 2
}
projectedPopulations

let bigProjections = projectedPopulations.filter {
    (projection: Int) -> Bool in
    return projection > 4000
}
bigProjections
```

跟上面一样，这里用了尾部闭包语法。这个闭包接受人口估算值作为参数，返回布尔值表示这个估算值是否通过了测试。在闭包内，我们检查估算值是否大于4000并返回结果。把通过测试的值放入`bigProjections`数组。只有两个估算值通过了测试，所以`bigProjections`包含4042和4314。

3. `reduce(_:_:)`

假设Knowhere的镇长要求城市规划者提供镇子的人口数。估算现在数据是分散在数组中的，城市规划者如何得到总数呢？`reduce(_:_:)`提供了很好的方法来完成这个任务。我们可以对数组调用`reduce(_:_:)`，其任务是把一组数据约简成一个值并返回。

代码清单13-18　把数组约简成一个值

```
...
let bigProjections = projectedPopulations.filter {
    (projection: Int) -> Bool in
    return projection > 4000
}
bigProjections

let totalProjection = projectedPopulations.reduce(0) {
    (accumulatedProjection: Int, precinctProjection: Int) -> Int in
    return accumulatedProjection + precinctProjection
}
totalProjection
```

`reduce(_:_:)`的第一个参数是一个可以在一开始进行累加的初始值（或者其他值）。第二个参数是一个闭包，定义了如何合并容器中的值。（注意这里用了尾部闭包语法。）这里要做的只是累加`projectedPopulations`数组中的估算值，所以初始值是0。闭包接受两个`Int`类型的参数：`accumulatedProjection`和`precinctProjection`。在`reduce(_:_:)`遍历数组时就会把这两个值累加。当函数结束运行时，`totalProjection`等于10 844。

13.8　青铜挑战练习

在本章中，我们对容器内的元素进行排序，并返回了一个新的整数升序数组。我们也可以对容器进行原地排序。修改对`volunteerCounts`进行排序的示例代码，改为按从小到大原地排序。

13.9　青铜挑战练习

在本章中，我们用`sorted(by:)`把容器元素从小到大排序。不过如果只是想对容器元素进行升序排序的话，其实还有一个更简单的方法。查看文档找到这个方法，并把它用在上面的练习中。

13.10　黄金挑战练习

用本章学到的知识简化上面的`reduce(_:_:)`调用。这段代码可以显著简化：应该可以用一行代码实现。完成后，再看看其他的高阶函数示例代码并进行练习。

Part 4

第四部分

枚举、结构体和类

第四部分会介绍大量新工具和新概念。我们会给项目增加特性，不过之后这些项目还会变化。这就跟实际写代码一样：有时候开始时用一种解决方案写应用，当有了更好的模式或者特性发生变化后就得修改代码。这并不代表一开始的代码或工具不好，只是说明它们在其他环境下更好。项目会进化和发展，某种情况下的完美决定在需求发生变化后也许就不能解决问题了。面对变化时灵活应对是“行走江湖”的必备能力。

第 14 章

枚　　举

一路学习到这里，你已经见过Swift提供的所有内建类型了，比如整数、字符串、数组和字典等。下面几章会展示这门语言创建自定义类型的能力。本章关注的重点是枚举（enumeration或者enum）。枚举能让你创建属于明确定义的几种情形之一的实例。如果你用过其他语言的枚举，那么本章的内容对你来说应该不陌生；但是Swift的枚举还有一些区别于其他语言的高级特性。

14.1　基本枚举

创建一个新的playground，命名为Enumerations。定义一个表示文本对齐方式的枚举，如代码清单14-1所示。

代码清单14-1　定义枚举

```
import Cocoa

var str = "Hello, playground"

enum TextAlignment {
    case left
    case right
    case center
}
```

定义枚举的方式是在enum关键字后跟枚举的名字。左花括号（{）表示枚举体的开始。枚举体至少包含一个case语句，表示枚举的可能值；这里有三个。现在枚举的名字（这里是TextAlignment）已经可以用作类型了，就像Int和String等我们已经用过的类型一样。

类型的名字（当然也包括枚举）通常以大写字母开头；如果要用到多个单词，就用驼峰式大小写命名：UpperCamelCasedType。变量、函数以及枚举的成员以小写字母开头；如果需要的话也用驼峰式大小写命名：lowerCamelCasedName。

因为枚举会声明新类型，所以现在可以创建这个类型的实例了，如代码清单14-2所示。

代码清单14-2　创建TextAlignment的实例

```
...
enum TextAlignment {
    case left
```

```
    case right
    case center
}

var alignment: TextAlignment = TextAlignment.left
```

尽管TextAlignment是自定义类型，编译器还是能够推断alignment的类型。因此可以省略alignment的显式类型声明，如代码清单14-3所示。

代码清单14-3 利用类型推断

```
...
var alignment: TextAlignment = TextAlignment.left
```

编译器对枚举的类型推断能力不仅限于变量声明。如果一个变量已经明确是某个特定的枚举类型，就可以在给变量赋值时省略枚举名，如代码清单14-4所示。

代码清单14-4 推断枚举类型

```
...
var alignment = TextAlignment.left
alignment = .right
```

注意，第一次创建alignment变量时必须指定枚举名和值，因为既要声明alignment的类型，还要把它初始化。下一行可以省略类型，只要给alignment再赋一个属于该类型的其他值即可。在传递枚举给函数或比较枚举时可以省略枚举类型，如代码清单14-5所示。

代码清单14-5 在比较枚举值时利用类型推断

```
...
alignment = .right

if alignment == .right {
    print("we should right-align the text!")
}
```

用if语句可以比较枚举值，不过通常使用switch语句处理。用switch将对齐信息用人类可读的方式打印出来，如代码清单14-6所示。

代码清单14-6 切换到switch

```
...
alignment = .right

if alignment == .right {
    print("we should right-align the text!")
}
switch alignment {
case .left:
    print("left aligned")

case .right:
    print("right aligned")
```

```
case .center:
    print("center aligned")
}
```

回忆第5章的内容，所有的switch语句必须被全覆盖。因此，第5章的switch语句有default分支。对枚举值用switch时就没这个必要了：编译器知道枚举的所有可能值。如果每一种可能值都有对应的分支，那么switch就被全覆盖了。

对枚举类型使用switch时也可以用default分支，如代码清单14-7所示。

代码清单14-7　把居中对齐作为默认选项

```
...
switch alignment {
case .left:
    print("left aligned")

case .right:
    print("right aligned")

case .center:
default:
    print("center aligned")
}
```

这段代码可以运行，不过我们建议在对枚举类型用switch时避免使用default分支，因为用默认分支不那么"面向未来"。假设后来想再为两端对齐文本增加一种对齐方式，如代码清单14-8所示。

代码清单14-8　增加成员值

```
...
enum TextAlignment {
    case left
    case right
    case center
    case justify
}

var alignment = TextAlignment.leftjustify
alignment = .right
...
```

注意，代码尽管还能运行，但是会输出错误的值。alignment变量设置为justify，但是switch语句打印了center aligned。这就是默认分支不面向未来的意思：这样做会增加未来修改代码的复杂度。

把switch改回原来的样子，如代码清单14-9所示。

代码清单14-9　回到显式的分支

```
...
switch alignment {
case .left:
```

```
    print("left aligned")

case .right:
    print("right aligned")

default:
case .center:
    print("center aligned")
}
```

现在如果出错的话，程序不会运行，更不会打印出错的信息，而是会产生一个编译时错误，告诉你switch语句没有被全覆盖。故意产生编译时错误可能很奇怪，但是在这里确实是有用的。

如果在对枚举用switch的同时使用default分支，那么switch语句会被永远全覆盖，从而通过编译器的检查。如果在枚举中增加了一个成员值但是没有更新switch语句，那么switch语句在遇到新的成员值时会跳到default。代码能编译通过，但不是你想要的结果，正如我们在代码清单14-8中看到的那样。

通过在switch中列出枚举的每个成员值可以确保当我们给枚举增加成员值时编译器能协助我们发现代码中所有必须更新的地方。这就是此处的情况：编译器告诉你switch语句没有包含枚举中定义的所有成员值。

我们来修复这个问题，如代码清单14-10所示。

代码清单14-10　包含所有成员值

```
...
switch alignment {
case .left:
    print("left aligned")

case .right:
    print("right aligned")

case .center:
    print("center aligned")

case .justify:
    print("justified")
}
```

14.2　原始值枚举

如果你用过C或C++中的枚举，会惊讶地发现Swift的枚举没有底层的整型。不过利用Swift称为原始值（raw value）的特性就能得到同样的效果。想让文本对齐枚举使用整型原始值，需要把枚举的声明改成如代码清单14-11所示的形式。

代码清单14-11　使用原始值

```
...
enum TextAlignment: Int {
```

14

```
        case left
        case right
        case center
        case justify
    }
    ...
```

为TextAlignment指定原始值类型会给每个成员设置一个该类型（Int）的原始值。整数原始值的默认行为是：第一个成员的原始值是0，第二个是1，以此类推。打印一些插值字符串来确认这一点，如代码清单14-12所示。

代码清单14-12　确认原始值

```
...
var alignment = TextAlignment.justify

print("Left has raw value \(TextAlignment.left.rawValue)")
print("Right has raw value \(TextAlignment.right.rawValue)")
print("Center has raw value \(TextAlignment.center.rawValue)")
print("Justify has raw value \(TextAlignment.justify.rawValue)")
print("The alignment variable has raw value \(alignment.rawValue)")
...
```

也可以不用原始值的默认行为。需要的话，可以给每个成员指定原始值，如代码清单14-13所示。

代码清单14-13　指定原始值

```
...
enum TextAlignment: Int {
    case left    = 20
    case right   = 30
    case center  = 40
    case justify = 50
}
...
```

原始值枚举有什么用？使用原始值的最常见原因莫过于需要存储或传输枚举。用rawValue可以把枚举变量转化成原始值，而不需要写一个函数变换它。

这带来了另一个问题：如果拿到一个原始值，如何将其转化回枚举类型呢？每个带原始值的枚举类型都可以用rawValue参数创建，并返回可空枚举，如代码清单14-14所示。

代码清单14-14　把原始值转化回枚举类型

```
...
print("Justify has raw value \(TextAlignment.justify.rawValue)")
print("The alignment variable has raw value \(alignment.rawValue)")

// 创建一个原始值
let myRawValue = 20

// 尝试将原始值转化为TextAlignment
```

```
if let myAlignment = TextAlignment(rawValue: myRawValue) {
    // 转化成功
    print("successfully converted \(myRawValue) into a TextAlignment")
} else {
    // 转化失败
    print("\(myRawValue) has no corresponding TextAlignment case")
}
...
```

这里发生了什么？首先有一个整型变量myRawValue，然后我们尝试用TextAlignment(rawValue:)将原始值转化为TextAlignment。因为TextAlignment(rawValue:)的返回值是TextAlignment?，所以可以用可空实例绑定来判断返回的是TextAlignment值还是nil。

这里用的原始值对应TextAlignment.left，所以转化成功了。试着把myRawValue改成不存在的原始值，看一下无法转化时的消息，如代码清单14-15所示。

代码清单14-15　试一下不存在的值

```
...
let myRawValue = 20 100
...
```

图14-1显示了else块的执行。

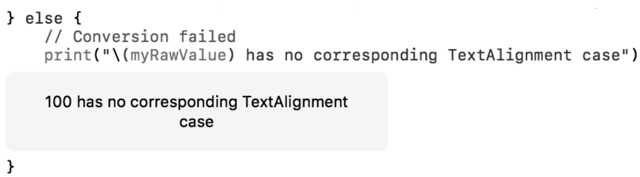


图14-1　TextAlignment转化失败的结果

到目前为止，我们用的都是整型原始值。Swift支持一系列类型，包括所有的内建数值类型和字符串。创建一个新的枚举，用String作为原始值类型，如代码清单14-16所示。

代码清单14-16　创建带字符串原始值的枚举

```
...
enum ProgrammingLanguage: String {
    case swift      = "swift"
    case objectiveC = "objective-c"
    case c          = "c"
    case cpp        = "c++"
    case java       = "java"
}

let myFavoriteLanguage = ProgrammingLanguage.swift
print("My favorite programming language is \(myFavoriteLanguage.rawValue)")
```


第一次用整型原始值的时候可以不指定具体的值——编译器会自动把第一个成员设置为0，第二个设置为1，以此类推。这里为每个成员指定了对应的字符串原始值，但也不是必需的：如

果省略原始值，Swift会使用成员本身的名字。修改ProgrammingLanguage，删除和成员名一样的原始值，如代码清单14-17所示。

代码清单14-17　使用默认的字符串原始值

```
...
enum ProgrammingLanguage: String {
    case swift       = "swift"
    case objectiveC = "objective-c"
    case c           = "c"
    case cpp         = "c++"
    case java        = "java"
}

let myFavoriteLanguage = ProgrammingLanguage.swift
print("My favorite programming language is \(myFavoriteLanguage.rawValue)")
```

你对Swift的爱并没有变化。

14.3　方法

方法是和类型关联的函数。有些语言的方法只能和类（第15章会讨论）关联。在Swift中，方法可以和枚举关联。创建一个新枚举代表电灯泡的状态，如代码清单14-18所示。

代码清单14-18　电灯泡可以开关

```
...
enum Lightbulb {
    case on
    case off
}
```

你可能想知道电灯泡的温度。（为简单起见，假设电灯泡打开后会马上变热，关掉后会马上降低到环境温度。）增加一个计算表面温度的方法，如代码清单14-19所示。

代码清单14-19　实现获取温度的方法

```
...
enum Lightbulb {
    case on
    case off

    func surfaceTemperature(forAmbientTemperature ambient: Double) -> Double {
        switch self {
        case .on:
            return ambient + 150.0

        case .off:
            return ambient
        }
    }
}
```

这段代码在Lightbulb枚举的内部增加了一个函数。因为函数定义在枚举内部，所以它是跟Lightbulb类型关联的方法。我们称之为Lightbulb的方法。这个函数看起来只接受一个参数（ambient），但因为它是方法，所以还接受一个隐式参数self，类型是Lightbulb。所有的Swift方法都有self参数，用来访问对应的实例（该方法在这个实例上调用）——在本例中就是Lightbulb实例。

创建一个变量代表电灯泡，调用新实现的方法，如代码清单14-20所示。

代码清单14-20 打开电灯泡

```
...
enum Lightbulb {
    case on
    case off

    func surfaceTemperature(forAmbientTemperature ambient: Double) -> Double {
        switch self {
        case .on:
            return ambient + 150.0

        case .off:
            return ambient
        }
    }
}

var bulb = Lightbulb.on
let ambientTemperature = 77.0

var bulbTemperature = bulb.surfaceTemperature(forAmbientTemperature:
                      ambientTemperature)
print("the bulb's temperature is \(bulbTemperature)")
```

首先创建一个Lightbulb类型的实例bulb。有了某个类型的实例，就可以利用语法instance.methodName(parameters)对这个实例调用方法。当你在此调用bulb.surfaceTemperature(forAmbientTemperature: ambientTemperature)时，就是在用这种语法调用方法。变量bulb是Lightbulb的实例，surfaceTemperature(forAmbientTemperature:)是所调用方法的名字，ambientTemperature是传递给方法的参数。这个方法调用的结果是Double，保存在bulbTemperature变量中。最后，把电灯泡的温度打印到控制台。

另一个有用的方法是开关电灯泡。要开关电灯泡，需要修改self的状态使之从on到off或者从off到on。试着增加一个toggle()方法，这个方法不需要参数也不返回任何东西，如代码清单14-21所示。

代码清单14-21 尝试开关


```
...
enum Lightbulb {
    case on
    case off
```

```
    func surfaceTemperature(forAmbientTemperature ambient: Double) -> Double {
        switch self {
        case .on:
            return ambient + 150.0

        case .off:
            return ambient
        }
    }

    func toggle() {
        switch self {
        case .on:
            self = .off

        case .off:
            self = .on
        }
    }
}
...
```

输入这些代码后，编译器会产生错误，告诉你不能在方法内对`self`赋值。在Swift中，枚举是值类型（value type），而值类型的方法不能对`self`进行修改（第15章会详细讨论值类型）。如果希望值类型的方法能修改`self`，需要标记这个方法为`mutating`。在代码中加上这个标记，如代码清单14-22所示。

代码清单14-22　标记`toggle()`方法为`mutating`

```
...
    mutating func toggle() {
        switch self {
        case .on:
            self = .off

        case .off:
            self = .on
        }
    }
...
```

现在可以开关电灯泡并看到关掉时的温度了，如代码清单14-23所示。

代码清单14-23　关掉电灯泡

```
...
var bulbTemperature = bulb.surfaceTemperature(forAmbientTemperature:
                        ambientTemperature)
print("the bulb's temperature is \(bulbTemperature)")

bulb.toggle()
bulbTemperature = bulb.surfaceTemperature(forAmbientTemperature: ambientTemperature)
print("the bulb's temperature is \(bulbTemperature)")
```

14.4 关联值

到目前为止,用枚举做的事情都是一类:定义一些静态的成员值来枚举可能的值或状态。Swift还提供了一种强大的枚举：带关联值的成员。关联值能让你把数据附在枚举实例上；不同的成员可以有不同类型的关联值。

创建一个枚举用来记录一些基本图形的尺寸。每种图形有不同的属性。要表示正方形，只需要一个值（边长）。要表示长方形，则需要两个值：宽和高。如代码清单14-24所示。

代码清单14-24　设置ShapeDimensions

```
...
enum ShapeDimensions {
    // 正方形的关联值是边长
    case oquare(side: Double)

    // 长方形的关联值是宽和高
    case rectangle(width: Double, height: Double)
}
```

这里定义了一个新的枚举类型ShapeDimensions,它有两个成员。square的关联值是(side: Double)类型。rectangle的关联值是(width:Double, height:Double)类型。两者都是命名元组（第一次出现在第12章）。

要创建ShapeDimensions的实例，必须指定成员和相应的关联值，如代码清单14-25所示。

代码清单14-25　创建图形

```
...
enum ShapeDimensions {
    // 正方形的关联值是边长
    case square(side: Double)

    // 长方形的关联值是宽和高
    case rectangle(width: Double, height: Double)
}

var squareShape = ShapeDimensions.square(side: 10.0)
var rectShape = ShapeDimensions.rectangle(width: 5.0, height: 10.0)
```

这里创建了一个边长是10单位的正方形，还有5单位×10单位的长方形。

用switch语句可以解开并使用关联值。给ShapeDimensions增加一个计算图形面积的方法，如代码清单14-26所示。

代码清单14-26　利用关联值计算面积

```
...
enum ShapeDimensions {
    // 正方形的关联值是边长
    case square(side: Double)
```

14

```
    // 长方形的关联值是宽和高
    case rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
        case let .square(side: side):
            return side * side

        case let .rectangle(width: w, height: h):
            return w * h }
        }
    }
}
...
```

在area()的实现代码中，我们对self用了switch，就跟本章前面一样。在这里，switch的分支利用了Swift的模式匹配（pattern matching）把self的关联值绑定到新变量上。

对实例调用area()方法来看一下实际效果，如代码清单14-27所示。

代码清单14-27　计算面积

```
...
var squareShape = ShapeDimensions.square(side: 10.0)
var rectShape = ShapeDimensions.rectangle(width: 5.0, height: 10.0)

print("square's area = \(squareShape.area())")
print("rectangle's area = \(rectShape.area())")
```

不是所有的枚举成员都必须有关联值。比如要增加一个point成员。由于几何点不存在尺寸，增加point并不需要关联值类型。

增加point并更新area()方法来计算其面积，如代码清单14-28所示。

代码清单14-28　设置Point

```
...
enum ShapeDimensions {
    // 点没有关联值——它没有尺寸
    case point

    // 正方形的关联值是边长
    case square(side: Double)

    // 长方形的关联值是宽和高
    case rectangle(width: Double, height: Double)

    func area() -> Double {
        switch self {
        case .point:
            return 0

        case let .square(side: side):
            return side * side
```

```
            case let .rectangle(width: w, height: h):
                return w * h }
            }
        }
    }
    ...
```

现在创建一个点的实例，确认area()像预期那样工作，如代码清单14-29所示。

代码清单14-29　点的面积是多少

```
...
var squareShape = ShapeDimensions.square(side: 10.0)
var rectShape = ShapeDimensions.rectangle(width: 5.0, height: 10.0)
var pointShape = ShapeDimensions.point

print("square's area = \(squareShape.area())")
print("rectangle's area = \(rectShape.area())")
print("point's area = \(pointShape.area())")
```

14.5　递归枚举

现在我们知道了如何把关联值附到枚举成员上。这引出了一个有趣的问题：枚举能给成员附上自己类型的关联值吗？（可能还有一个问题：你为什么想这么做？）

在计算机科学中，树是很常见的数据结构。大部分分层数据天然可以用树表示。假设有一份族谱包含了人（树的“节点”）和祖先关系（树的“边”）。遇到不认识的祖先时，族谱树的分支就到头了，如图14-2所示。

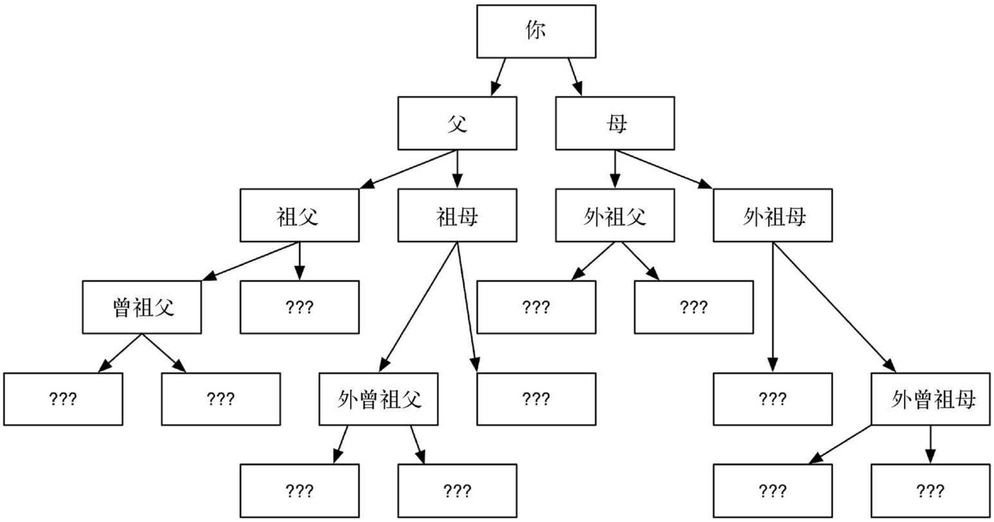


图14-2　族谱

为族谱建模很难，因为对于某个人的父母，你可能一个也不认识，也可能只认识其中之一，还有可能两个都认识。如果认识一个或两个都认识，还需要记录他们的祖先。试着创建一个枚举用来尽量完整地构建族谱，如代码清单14-30所示。

代码清单14-30　尝试定义FamilyTree

```
...
enum FamilyTree {
    case noKnownParents
    case oneKnownParent(name: String, ancestors: FamilyTree)
    case twoKnownParents(fatherName: String, fatherAncestors: FamilyTree,
                         motherName: String, motherAncestors: FamilyTree)
}
```

输入以上代码后，Xcode会报错并建议修复：“Recursive enum ‘FamilyTree’ is not marked ‘indirect’”。FamilyTree是*递归*的，因为其成员的关联值类型还是FamilyTree。不过，为什么编译器要管枚举是不是递归的呢？

要回答这个问题，需要理解枚举幕后的工作原理。Swift编译器必须知道程序中每种类型的每个实例占据多少内存空间。你（通常）不需要操心这些，因为编译器在构建程序时能帮你计算出来。不过，枚举有点麻烦。

尽管一个枚举的实例可能会随着程序的运行变为其他成员值，但是编译器知道，同一时间它只会是一个成员值。于是，当编译器判断一个枚举实例需要多少内存的时候，会查看每种成员值并找出需要最多内存的那个。实例需要的内存就是这么多（再加上一点额外的空间，编译器用其来记录当前是哪个成员值）。

回顾一下枚举ShapeDimensions。point成员没有关联数据，不需要额外的内存。square有Double关联值，需要Double的内存空间（8字节）。rectangle成员有两个关联的Double，需要16字节内存。ShapeDimensions实例的实际大小是17字节：足以装下rectangle的空间；如果需要的话，还有1字节记录该实例现在是哪个成员值。

现在考虑枚举FamilyTree。oneKnownParent需要多少内存？足以装下String的内存外加FamilyTree的内存。看到问题了吗？如果编译器不知道FamilyTree多大，就不知道FamilyTree多大。从另一个角度看，FamilyTree需要无限的内存！

为了解决这个问题，Swift引入了一个间接层。我们不再直接判断oneKnownParent需要多少内存（这样会绕回无限递归），而是用关键字indirect告诉编译器把枚举的数据放到一个指针指向的地方。我们在本书中不过多讨论指针，因为Swift不会让你直接接触指针。在本例中，除了让FamilyTree在幕后用指针之外不需要做任何事。之后就可以为族谱建模了，如代码清单14-31所示。

代码清单14-31　正确的FamilyTree

```
...
indirect enum FamilyTree {
    case noKnownParents
    case oneKnownParent(name: String, ancestors: FamilyTree)
```

```
    case twoKnownParents(fatherName: String, fatherAncestors: FamilyTree,
                         motherName: String, motherAncestors: FamilyTree)
}
```

指针是怎么解决“无限内存”问题的？编译器现在知道应该保存一个指向关联数据的指针，把数据放在内存中的其他地方而不是让FamilyTree的实例有足够的空间放下数据。现在FamilyTree的实例在64位架构下有8字节，即一个指针的长度。

值得注意的是，其实不需要把整个枚举标记为间接：也可以把递归的成员单独标记为间接。现在这样修改一下，如代码清单14-32所示。

代码清单14-32 FamilyTree的间接分支

```
...
~~indirect~~ enum FamilyTree {
    case noKnownParents
    indirect case oneKnownParent(name: String, ancestors: FamilyTree)
    indirect case twoKnownParents(fatherName: String, fatherAncestors: FamilyTree,
                                  motherName: String, motherAncestors: FamilyTree)
}
```

现在编译器能接受FamilyTree了，创建一个实例来为Fred的族谱建模。Fred认识的祖先不多，这是好事，因为敲那么多代码创建FamilyTree的实例很累！

他认识自己的父母，所以需要使用twoKnownParents分支。他只认识父亲的母亲，所以对父亲的祖先用oneKnownParent。他不认识母亲的双亲，也不认识祖母的双亲，所以对这些祖先都要用noKnownParents，如代码清单14-33所示。

代码清单14-33 创建FamilyTree

```
...
let fredAncestors = FamilyTree.twoKnownParents(
    fatherName: "Fred Sr.",
    fatherAncestors: .oneKnownParent(name: "Beth", ancestors: .noKnownParents),
    motherName: "Marsha",
    motherAncestors: .noKnownParents)
```

上面的新代码可以用图14-3表示。

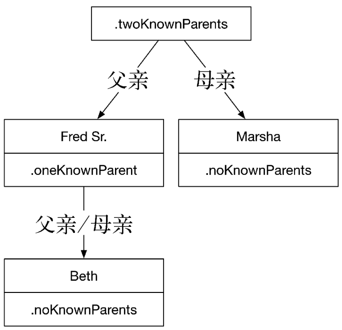


图14-3 Fred的族谱

`fredAncestors`是表示Fred族谱的递归枚举，树中每个节点表示同一个枚举的实例。如你所见，这种枚举能很好地为嵌套信息建模。

14.6 青铜挑战练习

给枚举`ShapeDimensions`增加一个`perimeter()`方法。这个方法用来计算图形的周长（所有边长之和）。确保处理所有的分支！

14.7 白银挑战练习

给枚举`ShapeDimensions`增加一个成员表示直角三角形。可以忽略三角形的朝向，只记录三条边的长度。增加一个成员会导致playground在`area()`方法中报错。修复这个错误。

第 15 章 结构体和类

结构体和类是构建应用这座大厦的支柱。它们提供重要的机制来为代码中要表达的事物建模。

下面的几章会从playground切换到命令行工具（command-line tool）。这个命令行工具的工程表示一个饱受怪兽侵扰的镇子。我们会用到结构体和类来为这些实体建模，并为其增加用来保存数据的属性，以及让这些实体完成一些任务的函数。

你将会看到，结构体和类有相似之处也有不同之处。在特定场景下应该用结构体还是类是个重要抉择。本章先从理解它们各自的优势开始，你会在第18章更好地了解它们各自的适用场景。

15.1 新工程

在欢迎窗口中点击Create a new Xcode project，创建一个新工程（如图15-1所示）。如果Xcode已经在运行了，点击File → New → Project...。

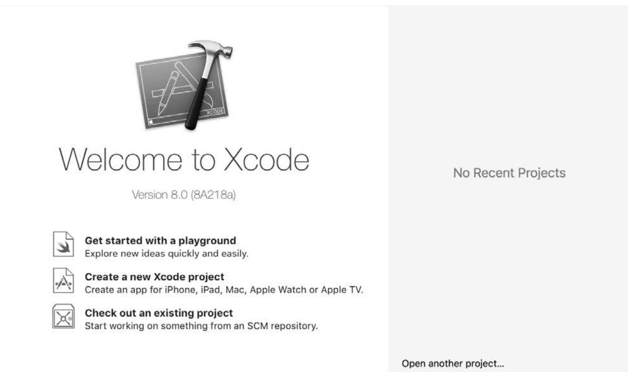


图15-1　欢迎窗口

接下来选择工程模版。模版用一组对某种应用通用的预设值和配置来设置工程。注意窗口的顶部有几个选项：iOS、watchOS、tvOS、macOS和Cross-platform。选择macOS，接着在窗口的Application区域选择Command Line Tool模版并点击Next（如图15-2所示）。这个模版会创建一个很基本的工程。

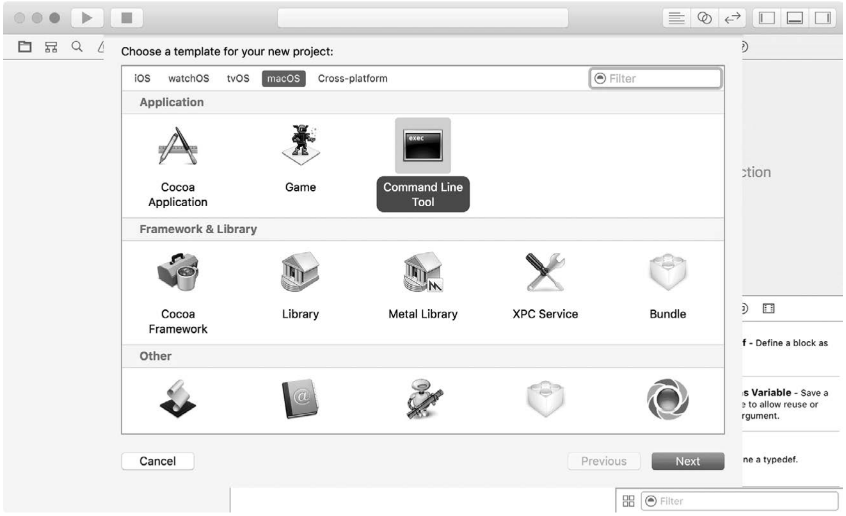


图15-2　选择模版

现在要选择工程的选项，包括名字（如图15-3所示）。在Product Name一栏填写MonsterTown，工程的Organization Name填写BigNerdRanch（如果你喜欢的话也可以填别的）。Organization Identifier使用反向域名表示法（reverse DNS）自动为你填好了。这个字符串会和Product Name合起来创建Bundle Identifier。当你准备好发布应用时，bundle ID用来在iTunes Connect中识别应用。（不用管Team那一栏，它是用来签名并发布应用的。）

在Language一栏选择Swift，点击Next。

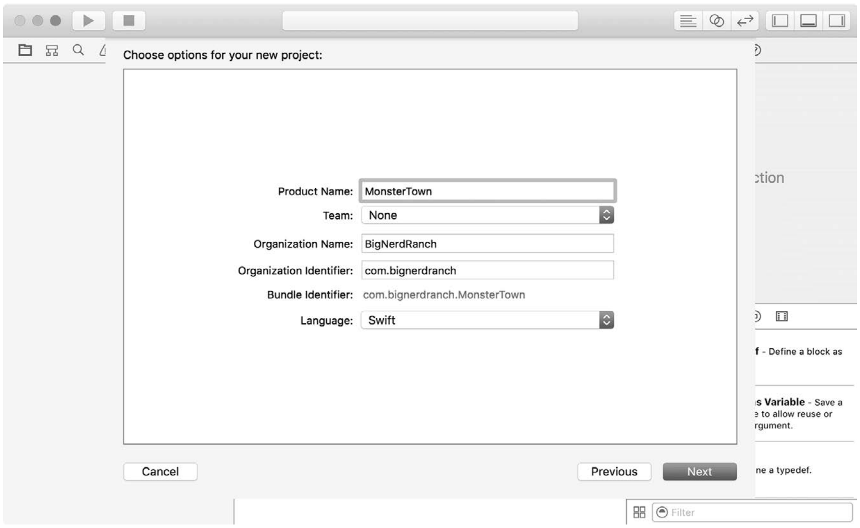


图15-3　给工程命名

最后，Xcode会询问在哪里保存工程。选择一个合适的本地路径并点击Create。

现在工程会在Xcode中打开，默认已经选中工程文件，如图15-4所示。这个文件用来管理应用的各种设置。比如，可以签名以便部署应用，也可以链接开发所需的框架，还有很多其他设置。

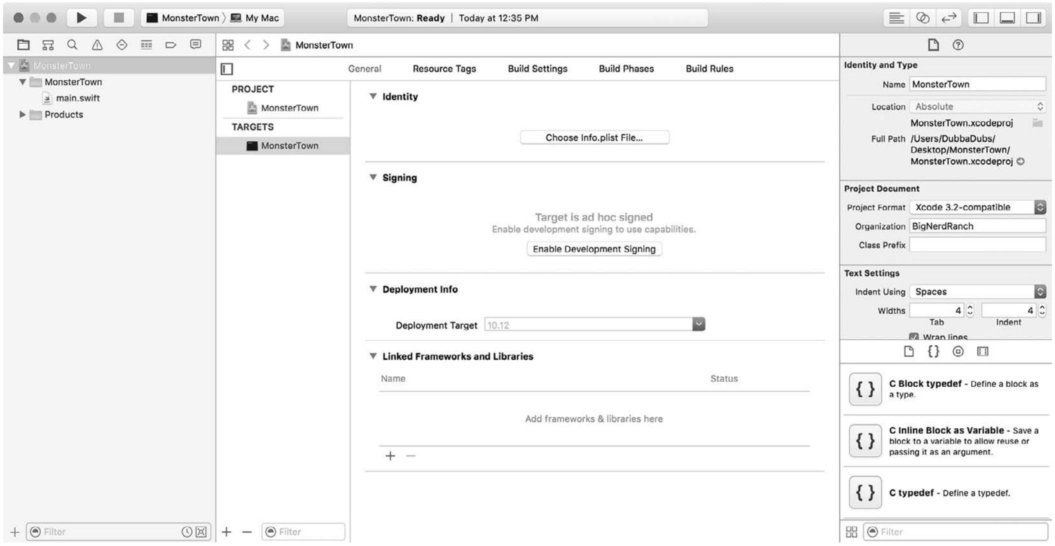


图15-4　工程文件

现在花些时间来看一下Xcode的窗口布局。图15-5提供了最重要的几个区域的概览。



图15-5　Xcode的布局

左边的面板是导航区（navigator area），提供了包含工程组织方式的几种视图。默认打开的视图是工程导航器（project navigator）。在工程导航器中有文件列表，目前其中只有main.swift。

中间是编辑区（editor area）。这里是添加、查看和编辑代码的地方。

右边是工具区（utilities area），提供一些获取更多信息的检查面板。比如文件检查面板（file inspector）能够显示文件的位置、名字等信息。

Xcode窗口的底部是调试区（debug area）。遇到问题时用这个区域调试代码。

窗口顶部是工具栏，你会用到Play和Stop按钮来运行和停止程序。工具栏的最右边还有三个按钮，分别用来显示和隐藏导航区、调试区和工具区。

注意Xcode为应用创建了一个main.swift文件（如图15-6所示）

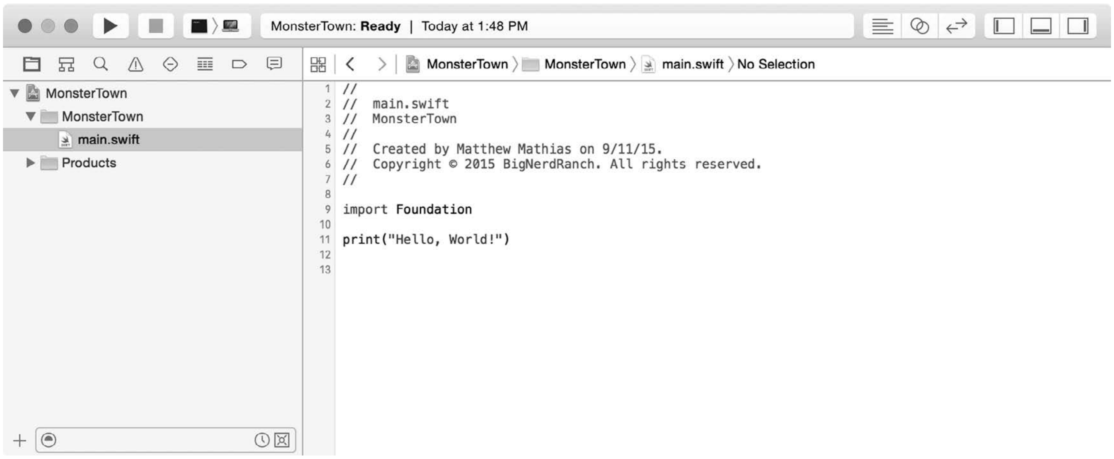


图15-6　main.swift

对于命令行工具，main.swift表示程序的入口点。main.swift通常包含“顶层”代码，也就是不包含在任何函数中或不在其他类型（比如结构体或类）中定义的代码。这个文件中的代码是顺序执行的：从上到下。

因为main.swift是程序开始运行的地方，所以这个文件中的代码通常用来做初始化工作。我们将在其他文件中定义类型，在main.swift中创建这些类型的实例。比如，创建一个Town.swift文件保存结构体`Town`的定义，然后在main.swift中创建`Town`的实例。

类型经常在自己的文件中定义，这样有利于组织应用的源代码。这种方法可以让你更容易找到并调试代码。

注意main.swift中已经有了下面几行代码：

```
import Foundation

print("Hello, World!")
```

`import Foundation`为main.swift带来了`Foundation`框架（framework）。这个框架包含一组

主要用来和Objective-C交互的类。后面会忽略这行代码，除非是在代码展示或明确需要用到这个框架提供的某个类型时。`print("Hello, World!")`这行代码看起来很熟悉，它会把字符串`Hello, World!`打印到控制台。

可以通过以下几种方式构建并运行程序：

- 点击屏幕顶部工具栏中的Product（如图15-7所示），然后选择Run；
- 在键盘上按下Command-R；
- 点击左上角的三角形Play按钮。



图15-7　Xcode工具栏

运行程序后，`Hello, World!`会被打印到控制台，一起打印出来的还有编译器关于程序结束状态的信息。这很棒，但是你之前已经见到过了。现在来创建自定义的结构体和类，使程序变得更有趣。开始之前，先删掉`print("Hello, World!")`，这行代码没用了（如代码清单15-1所示）。

代码清单15-1　删除“Hello, World!”（main.swift）

```
import Foundation

print("Hello, World!")
```

15.2　结构体

结构体（struct）是把相关数据块组合在一起放在内存中的一种类型。当需要把数据组合为一种通用类型时就可以用结构体。比如，在MonsterTown中创建结构体`Town`来为一个有怪兽袭扰问题的镇子建模。

把`Town`实现为结构体可以把它的数据封装为一种类型，把它的定义放在自己的文件中可以方便我们找到实现代码。在前几章中，我们在playground中模拟了一个镇子。这个例子相对来说比较小，所以放在playground中也没什么问题。用playground做快速的原型代码开发很好，但是距离真实的应用开发工程还很远。把镇子的定义用它自己的类型封装起来会更好。

点击File → New → File...给工程添加一个新文件，也可以用快捷键Command-N。一个如图15-8所示的新窗口会出现，提示你为新文件选择模版。选择顶部的macOS，再选择Source区域的Swift File并点击Next。

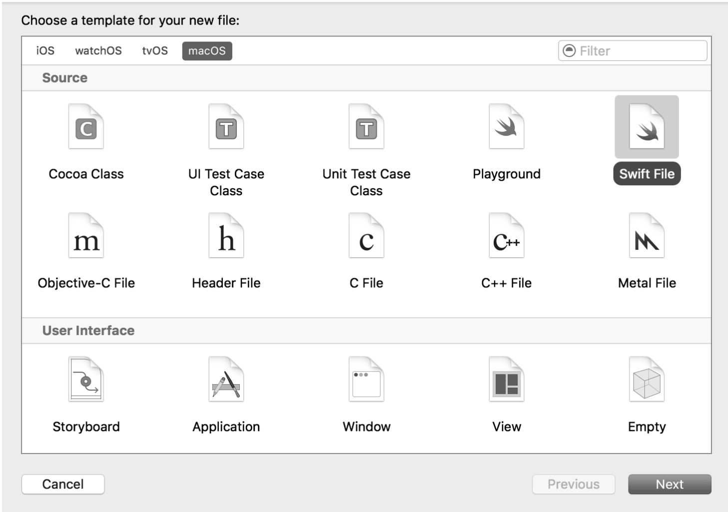


图15-8　添加Swift文件

接下来Xcode会要求你提供新文件的名字和位置。把这个文件命名为Town，确保勾选下面的复选框以便将其加入MonsterTown（如图15-9所示）。点击Create。

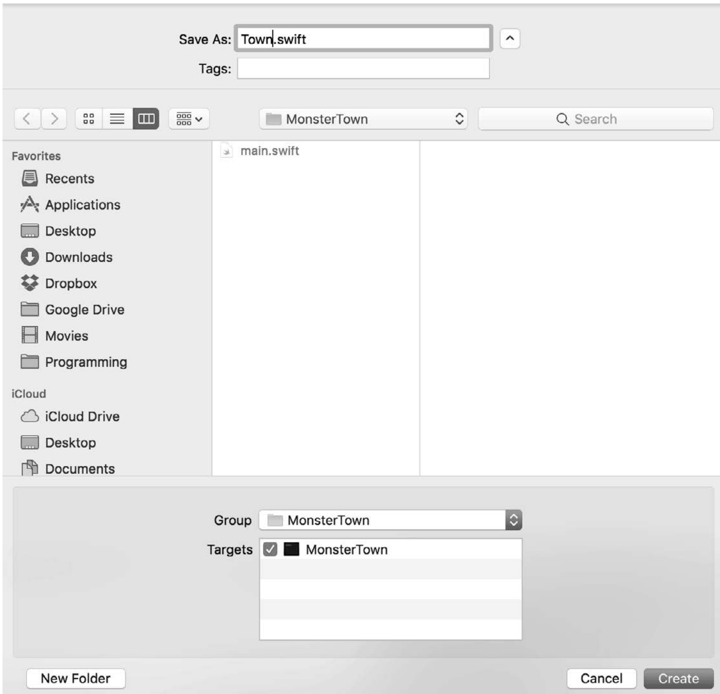


图15-9　Town.swift

新建的文件会自动打开。（如果没有，在导航区选择Town.swift。）打开后，你会发现除了顶部的一些注释信息和`import Foundation`行，这个文件几乎是空的。

先声明结构体`Town`，如代码清单15-2所示。

代码清单15-2 声明结构体（Town.swift）

```
import Foundation

struct Town {

}
```

关键字`struct`表示结构体声明，本例中就是`Town`。定义结构体行为的代码放在花括号（`{}`）之间。比如说，可以给结构体添加一些变量，用来保存镇子特征的数据，从而帮助建模。

严格来说，这些变量叫作*属性*（property），这是下一章的主题。属性可以是变量或常量，用之前见过的`var`和`let`关键字声明。为结构体添加一些属性，如代码清单15-3所示。

代码清单15-3 添加属性（Town.swift）

```
struct Town {
    var population = 5_422
    var numberOfStoplights = 4
}
```

这里，我们给`Town`添加了两个属性：`population`和`numberOfStoplights`。两个属性都是可变的——这很合理，因为镇子的人口和红绿灯数量会随着时间变化。为简单起见，这些属性也有默认值。当一个结构体`Town`的实例被创建时，默认有5422的人口和4个红绿灯。

切换到main.swift文件，创建一个`Town`的新实例，看一下结构体的实际行为，如代码清单15-4所示。

代码清单15-4 创建Town的实例（main.swift）

```
var myTown = Town()
print("Population: \(myTown.population),
       number of stoplights: \(myTown.numberOfStoplights)")
```

（注意，因为书页尺寸的限制，上面`print()`中的代码分成了两行。如果原封不动地像上面这样输入代码会出错。把`print()`写到一行可以避免这个问题。）

这段代码做了三件事情。

首先，输入类型名（这里是`Town`），后面跟上圆括号`()`，从而创建了`Town`类型的实例。用空的圆括号会调用`Town`的默认初始化方法（第17章会详细讨论初始化）。

其次，把新实例赋给变量`myTown`。

最后，用字符串插值把结构体`Town`的两个属性的值打印到控制台。注意访问属性的值要用点语法。比如，`myTown.population`会获取`myTown`实例的人口。

运行程序，输出是：`Population: 5422, number of stoplights: 4`（如图15-10所示）。

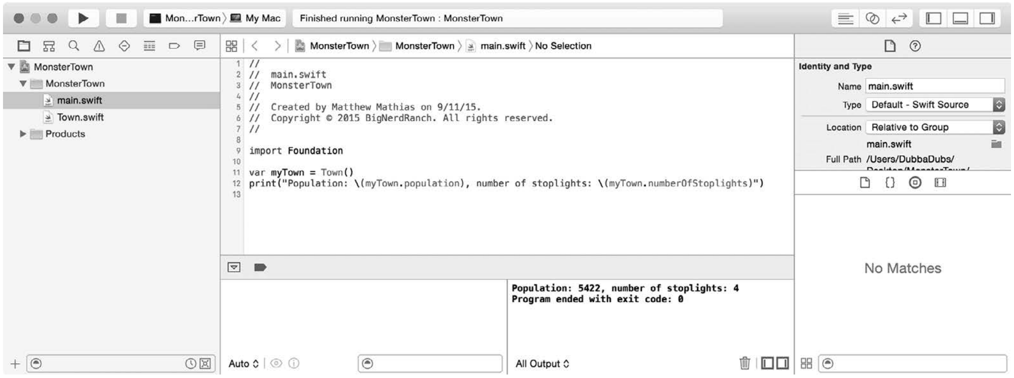


图15-10　描述myTown

15.3　实例方法

上面的`print()`函数是打印`myTown`描述信息的好办法。但是镇子应该知道如何描述自己。在结构体`Town`上创建一个函数来把属性的值打印到控制台。切换到Town.swift，添加如代码清单15-5所示的函数定义。

代码清单15-5　让Town描述自己（Town.swift）

```
struct Town {
    var population = 5_422
    var numberOfStoplights = 4

    func printDescription() {
        print("Population: \(population);
                number of stoplights: \(numberOfStoplights)")
    }
}
```

`printDescription()`是一个方法，因为它是跟特定类型关联的函数。（回忆第14章中关于方法的定义。）到目前为止，我们主要用到的函数都是全局函数（global function）。全局函数是没有定义在任何类型上的函数，所以也称为自由函数（free function）。

`printDescription()`不需要参数也没有返回值。它的目的是把镇子属性的描述打印到控制台。因为我们是对一个特定的`Town`实例调用这个方法的，所以`printDescription()`就是一个实例方法（instance method）。

要使用这个新的实例方法，需要对一个`Town`实例进行调用。切换回main.swift，把`print()`函数替换为新的实例方法，如代码清单15-6所示。

代码清单15-6　调用新的实例方法（main.swift）

```
var myTown = Town()
print("Population: \(myTown.population),
```

```
            ~~number of stoplights: \(myTown.numberOfStoplights)")~~
myTown.printDescription()
```

对实例调用函数要使用点语法：myTown.printDescription()。

运行程序，控制台的输出跟之前一样。

15.4　mutating 方法

用printDescription()方法可以很好地显示镇子当前的信息，但是如果需要一个修改镇子信息的函数呢？如果结构体的一个实例方法要修改结构体的属性，就必须标记为mutating。在Town.swift中，给Town类型添加一个mutating方法来增加镇子的人口，如代码清单15-7所示。

代码清单15-7　添加增加人口的mutating方法（Town.swift）

```
struct Town {
    var population = 5_422
    var numberOfStoplights = 4

    func printDescription() {
        print("Population: \(population);
              number of stoplights: \(numberOfStoplights)")
    }

    mutating func changePopulation(by amount: Int) {
        population += amount
    }
}
```

注意，实例方法changePopulation(by:)要用mutating关键字标记。跟第14章一样，这意味着方法可以修改结构体的值。结构体和枚举都是值类型（本章后面还会提到），需要在能修改实例属性的方法前加上mutating关键字。

这个方法有一个整型参数amount，用来增加镇子的人口：population += amount。切换到main.swift练习使用这个函数，如代码清单15-8所示。

代码清单15-8　增加人口（main.swift）

```
var myTown = Town()
myTown.changePopulation(by: 500)
myTown.printDescription()
```

跟之前一样，用点语法调用镇子的函数。如果编译并运行程序，你会看到myTown的人口增加了500，终端也输出了Population: 5922; number of stoplights: 4。

15.5　类

跟结构体类似，类可以用来为抽象成一个通用类型的相关数据建模。我们会在MonsterTown中用类为侵扰镇子的各种怪兽建模。类有几个重要的地方不同于结构体，本节会细致地讲解这些区别。

15.5.1　Monster 类

有了表示镇子的结构体，可以让事情变得更有趣一些了。不幸的是镇子有大批怪兽出没，这对于属性值而言可不是件好事。

创建一个新的Swift文件，命名为Monster。跟之前一样，点击File → New → File...或者按下Command-N，选择macOS下Source区域的Swift File模版。

这个文件会包含Monster类的定义。Monster类用来对怪兽的属性和袭击镇子的行为建模。先创建一个新类，如代码清单15-9所示。

代码清单15-9　创建怪兽（Monster.swift）

```
import Foundation

class Monster {

}
```

定义新类的语法和定义新结构体几乎完全一样。首先是关键字class，后面是新类的名字。跟之前一样，类的定义放在花括号之间。

出于继承（下一节会讨论）的考虑，我们用通用概念定义Monster类，如代码清单15-10所示。这意味着Monster类会描述怪兽的通用行为。后面会创建具有特定行为的各种怪兽。

代码清单15-10　定义Monster类（Monster.swift）

```
class Monster {
    var town: Town?
    var name = "Monster"

    func terrorizeTown() {
        if town != nil {
            print("\(name) is terrorizing a town!")
        } else {
            print("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

众所周知，怪兽擅长一件事：侵扰镇上的居民。Monster类有一个属性表示怪兽侵扰的镇子。回忆一下，当实例可能是nil时就要用可空类型。因为怪兽可能已经找到了一个镇子，也可能还没找到，所以town属性可空（Town?），而且初始值是nil。我们还创建了一个属性表示Monster的名字，并且为其赋了一个通用默认值。

接着定义方法terrorizeTown()的基本结构。我们会对Monster实例调用这个方法，表示怪兽在侵扰镇子。

注意我们会用if town != nil检查这个实例是否有town。如果有，那么terrorizeTown()会在控制台打印正在肆虐的怪兽名字。如果这个实例还没有镇子，那么这个方法会打印相应的信息。

因为每种怪兽侵扰镇子的方法各不相同，所以子类（subclass）会提供自己对这个函数的实现。我们会在下一节学习子类。

切换到main.swift，练习使用Monster类。添加这个类型的实例，给它设置一个镇子，然后调用terrorizeTown()方法，如代码清单15-11所示。

代码清单15-11 放出一只通用怪物（main.swift）

```
var myTown = Town()
myTown.changePopulation(by: 500)
myTown.printDescription()
let genericMonster = Monster()
genericMonster.town = myTown
genericMonster.terrorizeTown()
```

首先创建Monster的实例genericMonster。这个实例是一个常量，因为没必要修改。接着，把myTown赋给genericMonster的town属性。最后，对这个Monster实例调用terrorizeTown()方法。运行程序，控制台会打印Monster is terrorizing a town!。

15.5.2 继承

类的一个主要特性是继承（inheritance），而这是结构体没有的。继承是指由一个类（父类）定义另一个类（子类）的关系。子类继承父类的属性和方法。从某种意义上说，继承定义了类的系谱图。

1. Zombie子类

按之前创建Town.swift和Monster.swift的步骤创建一个Swift文件，命名为Zombie。

这个文件会保存表示僵尸的类的定义。Zombie类继承自Monster类。添加如代码清单15-12所示的类声明来学习如何定义子类。

代码清单15-12 创建僵尸（Zombie.swift）

```
import Foundation

class Zombie: Monster {
    var walksWithLimp = true

    override func terrorizeTown() {
        town?.changePopulation(by: -10)
        super.terrorizeTown()
    }
}
```

这个类定义了Zombie类型，它继承自Monster类型，Zombie后面的冒号和父类的名字（Monster）说明了这一点。继承自Monster意味着Zombie具备Monster的所有属性和方法，比如这里用到的属性town和方法terrorizeTown()。

Zombie还新增了一个布尔型属性walksWithLimp，其类型是从属性默认值true中推断出来的。

最后，Zombie重写了terrorizeTown()方法。重写方法意味着子类为父类定义的方法提供了自己的定义。注意关键字override的使用。重写方法时不用这个关键字会导致编译器报错。

图15-11展示了Zombie和Monster的关系。

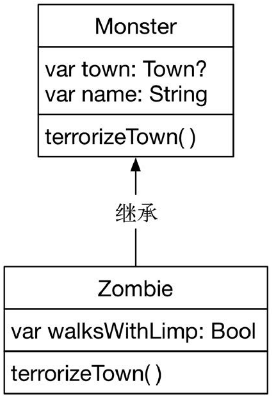


图15-11　Zombie的继承关系

Zombie继承了Monster类的属性town和name，也继承了terrorizeTown()方法，但是重写了这个方法，这就是图中两个类中都列出这个方法的原因。最后，Zombie添加了自己的属性：walksWithLimp。

注意代码清单15-12中super.terrorizeTown()那一行。super是一个前缀，用来访问父类的方法实现。在本例中，我们用super调用了Monster类对terrorizeTown()的实现。

super是基于继承思想的产物，在枚举、结构体等值类型上不可用。调用super可以从父类借用功能或重写父类的功能。

回忆一下，Zombie的属性town继承自Monster类，是可空类型Town?。我们需要确保：在对town调用任何方法之前，Zombie的实例要有一个可以侵扰的镇子。如何检查呢？

一种可能的解决方案是可空实例绑定。你可能想这么写：

```
if let terrorTown = town {
    // 侵扰镇子
}
```

在上面的代码中，如果Zombie实例有town，那么可空实例的值会被展开放进常量terrorTown。从这里开始，就可以侵扰这个镇子了，但是有一点要注意：结构体的本质决定了实例terrorTown和实例town不一样。为什么？因为包括结构体在内的值类型在传递时总是会被复制。

这样会出现问题，任何对terrorTown所做的修改都不会反映在Zombie实例的town属性上。除了这个限制，这段代码也可以更简洁。简单地说，这不是理想的解决方案。

第8章讲到过，可空链式调用能用一行代码完成这样的检查。代码的表达能力一样，而且更简洁。此外，还避开了上面提到的复制问题，因为我们会直接修改相关的那个Town实例。

我们已经在MonsterTown中用过可空链式调用了。回头看一下代码清单15-12，town?.changePopulation(by: -10)确保对town实例调用方法是安全的。如果可空town非空，就对这个实例调用方法changePopulation(by:)，而人口就会减少10。稍后，我们会用可空链式调用对僵尸的town调用printDescription()。

2. 禁止重写

有时候，我们希望禁止子类重写方法或属性。这样的需求在实际工作中很罕见，但是确实会出现。在这样的情况下，可以用关键字final让方法或属性不可重写。

举个例子，你不想让Zombie类型的子类重写terrorizeTown()方法。换句话说，Zombie的所有子类都用同样的方式侵扰镇子。在这个函数的声明上加上final关键字，如代码清单15-13所示。

代码清单15-13　禁止重写terrorizeTown()（Zombie.swift）

```
class Zombie: Monster {
    var walksWithLimp = true

    final override func terrorizeTown() {
        town?.changePopulation(by: -10)
        super.terrorizeTown()
    }
}
```

现在Zombie的子类无法重写terrorzieTown()方法了。创建Zombie的一个新子类ZombieBoss（跟之前一样，用新的Swift文件），然后试试重写terrorizeTown()方法，如代码清单15-14所示。

代码清单15-14　惹麻烦的僵尸王（ZombieBoss.swift）

```
import Foundation

class ZombieBoss: Zombie {
    override func terrorizeTown() {
        print("terrorizing town...")
    }
}
```

你会在重写terrorizeTown()方法的那一行看到如下错误：Instance method overrides a 'final' instance method。该错误表示我们不能重写terrorizeTown()，因为父类将其标记为了final。

继续往下，删除ZombieBoss.swift文件。这个文件以后用不到了。在工程导航区选择这个文件并按Delete键，在弹出窗口中选择Move to Trash。

3. 镇子出僵尸了

现在是练习使用Zombie类型的好时机。切换到main.swift文件，创建一个Zombie类的实例，如代码清单15-15所示。（注意，要删除打印镇子描述信息的代码，这样控制台的内容不会显得那么杂乱。还要删掉Monster的通用实例，它也没用了。）

代码清单15-15　谁害怕僵尸Fred（main.swift）

```
var myTown = Town()
myTown.changePopulation(by: 500)
myTown.printDescription()
let genericMonster = Monster()
genericMonster.town = myTown
genericMonster.terrorizeTown()
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printDescription()
```

首先创建Zombie类型的实例fredTheZombie，接着把已有的Town类型实例myTown赋给Zombie类型的属性town。到了这里，fredTheZombie就可以随意侵扰镇子了，它很乐意这么做。（至少是僵尸表现出的那么乐意。）

在fredTheZombie侵扰镇民之后，可以用printDescription()查看结果。之前讨论过，fredTheZombie的town属性是可空类型Town?，所以对其调用printDescription()方法前必须先展开。这可以用可空链式调用做到：fredTheZombie.town?。这行代码确保使用printDescription()之前fredTheZombie是有镇子的。

fredTheZombie侵扰完镇子后，控制台会输出：Population: 5912; number of stoplights: 4。

15.6　应该用哪种类型

用结构体还是用类，这是个难题。要回答这个问题，需要理解值类型和引用类型的区别。我们会在第18章讨论两者各自的特点，并提供合理利用这两种类型的指导。

15.7　青铜挑战练习

现在Zombie类型有个bug。如果一个Zombie实例侵扰了一个人口为0的镇子，其人口会降到–10。这个结果没有意义。修改Zombie类型的terrorizeTown()方法，使其只在镇子人口大于0的情况下才减少镇子人口。此外，如果要减少的人口大于当前人口，确保把人口置为0。

15.8　白银挑战练习

再创建一个Monster类型的子类Vampire。重写terrorizeTown()方法，使得每个Vampire实例侵扰镇子后，都会给Vampire类型的吸血鬼奴仆（vampire thrall）数组添加新的一员。这个吸血鬼奴仆数组默认应该是空的。侵扰镇子应该让镇子的人口减少1人。最后，在main.swift中练习使用Vampire类型。

15.9 深入学习：类型方法

本章定义了一些实例方法，我们对类型的实例调用这些方法。举个例子，terrorizeTown()是实例方法，可以对Monster类型的实例调用。还可以定义对类型本身进行调用的方法，我们称之为类型方法。类型方法对类型级别的信息很有用。

假设有一个结构体Square：

```
struct Square {
    static func numberOfSides() -> Int {
        return 4
    }
}
```

对于值类型，要声明类型方法需要用到static关键字。方法numberOfSides()只是简单地返回Square有几条边。

作为对比，类的类型方法要用class关键字标记。下面是一个Zombie类的类型方法，表示通用的僵尸流行语。

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    ...
}
```

想要使用类型方法，只要对类型本身调用就好了。

```
let sides = Square.numberOfSides() // 边数是4
let spookyNoise = Zombie.makeSpookyNoise() // 可怕的声音"Brains..."
```

把makeSpookyNoise()实现为类方法意味着子类可以用自己的实现覆盖这个方法。

```
class GiantZombie: Zombie {
    override class func makeSpookyNoise() -> String {
        return "ROAR!"
    }
}
```

这里的GiantZombie类继承了Zombie，并为类方法提供了自己的实现。但是如果你不想让子类继承某个类的类方法应该怎么办呢？可能你只想让僵尸都发出同一种可怕的声音。我们再来看看Zombie类。

```
class Zombie: Monster {
    static func makeSpookyNoise() -> String {
        return "Brains..."
    }

    ...
}
```

关键字static告诉编译器不要让子类为makeSpookyNoise()方法提供自己的实现。

也可以用`final class`代替`static`关键字，它们实现的功能一样。

```
class Zombie: Monster {
    final class func makeSpookyNoise() -> String {
        return "Brains..."
    }

    ...
}
```

类型方法可以使用给定类型的类型级别信息。这意味着类型方法可以调用其他类型方法，甚至可以使用类型属性；我们将在第16章讨论类型属性。不过要注意的是，类型方法不能调用实例方法，也不能用实例属性。这是因为实例不能在类型级别使用。

15.10　深入学习：mutating 方法

学习完本章后，你可能想了解`mutating`关键字。为什么修改结构体和枚举时需要这个关键字呢？从返回函数的函数开始研究会比较有用。

创建一个新的playground，命名为Mutating。在这个playground中添加一个简单的函数，返回一个问候字符串（如代码清单15-16所示）。

代码清单15-16　一个简单的问候函数

```
func greet(name: String, withGreeting greeting: String) -> String {
    return "\(greeting) \(name)"
}
```

`greet(name:withGreeting:)`函数接受两个参数：`name`和`greeting`。它会根据这两个参数构建一句问候语。这个函数用起来也很简单，如代码清单15-17所示。

代码清单15-17　使用`greet(name:withGreeting:)`

```
func greet(name: String, withGreeting greeting: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greet(name: "Matt", withGreeting: "Hello,")
print(personalGreeting)
```

现在，写一个会返回一个函数的新函数来根据名字问候一个人，如代码清单15-18所示。

代码清单15-18　从`greeting(forName:)`返回一个函数

```
func greet(name: String, withGreeting greeting: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greet(name: "Matt", withGreeting: "Hello,")
print(personalGreeting)
```

```
func greeting(forName name: String) -> (String) -> String {
    func greeting(_ greeting: String) -> String {
        return "\(greeting) \(name)"
    }
    return greeting
}
```

greeting(forName:)函数接受一个参数（字符串name）并返回一个函数。这个返回的函数本身接受一个字符串，表示问候语，并返回一个字符串表示对给定名字的问候。

greeting(forName:)的实现内部定义了嵌套函数greeting(_:)。这个嵌套函数的类型和greetingForName(_:)指定的类型一致——接受一个字符串并返回一个字符串。注意，我们用来自两个不同函数的参数greeting和name组成了个性化的问候语。

最后，返回greeting(_:)函数。

添加如代码清单15-19所示的代码来练习使用这个函数。

代码清单15-19　使用这个函数

```
func greet(name: String, withGreeting greeting: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greet(name: "Matt", withGreeting: "Hello,")
print(personalGreeting)

func greeting(forName name: String) -> (String) -> String {
    func greeting(_ greeting: String) -> String {
        return "\(greeting) \(name)"
    }
    return greeting
}

let greetMattWith = greeting(forName: "Matt")
let mattGreeting = greetMattWith("Hello,")
print(mattGreeting)
```

首先调用greeting(forName:)并传入我们要问候的人的名字（"Matt"），把结果赋给常量greetMattWith。greetMattWith持有和greeting(forName:)函数返回值类型一致的函数：接受一个字符串并返回一个字符串。我们指定的名字"Matt"通过greeting(forName:)函数返回的greeting(_:)函数的闭合作用域传递。

要给指定的名字制作个性化的问候语，可以调用greetMattWith(_:)函数并给其唯一的参数传入问候语（这里是"Hello,"）。把这个函数的结果赋给mattGreeting，然后打印到控制台。你会看到结果和之前一样。

你可能记得第13章讲到过Swift提供了一种直接返回闭包的方法。这个特性提供了一种不需要嵌套函数就可以实现greeting(forName:)的方法。把greeting(forName:)替换为greeting(_:)，这个新函数用了闭包而不是嵌套函数（如代码清单15-20所示）。

代码清单15-20　更紧凑的函数

```
func greet(name: String, withGreeting greeting: String) -> String {
    return "\(greeting) \(name)"
}

let personalGreeting = greet(name: "Matt", withGreeting: "Hello,")
print(personalGreeting)


func greeting(forName name: String) -> (String) -> String {
    func greeting(_ greeting: String) -> String {
        return "\(greeting) \(name)"
    }
    return greeting
}

let greetMattWith = greeting(forName: "Matt")
let mattGreeting = greetMattWith("Hello,")


func greeting(_ greeting: String) -> (String) -> String {
    return { (name: String) -> String in
        return "\(greeting) \(name)"
    }
}

let friendlyGreetingFor = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(mattGreeting)
```

注意，这个实现更加紧凑，而结果是一样的。

调用这个函数和之前的实现差不多。先调用greeting(_:)并向第一个参数传入一个字符串。把返回的函数赋给常量friendlyGreetingFor。接下来，调用friendlyGreetingFor并传入要问候的人名。

查看控制台，结果应该是一样的。

既然想起来了从函数返回函数，那么现在回到mutating关键字。创建一个结构体Person，如代码清单15-21所示。

代码清单15-21　创建Person

```
...
let friendlyGreetingFor = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(mattGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeTo(firstName: String, lastName: String) {
```

```
        self.firstName = firstName
        self.lastName = lastName
    }
}
```

这里没有我们不熟悉或者特殊的内容。Person结构体有一个人的姓氏和名字属性，还定义了一个mutating方法修改这些属性。

创建一个Person的实例，如代码清单15-22所示。

代码清单15-22 创建一个Person的实例

```
...
let friendlyGreetingFor = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(mattGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeTo(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

var p = Person()
```

这里也没有新内容，但是开始变得有趣起来。Swift的实例方法，也就是本章我们学习的这些，实际上是返回函数的类型级别的方法。输入如代码清单15-23所示的代码来看一下实际效果。

代码清单15-23 实例方法是返回函数的类型方法

```
...
let friendlyGreetingFor = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(mattGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeTo(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

var p = Person()
let changeName = Person.changeTo
```

我们可以访问Person结构体的changeTo(firstName:lastName:)方法。注意，不是调用方法（没有changeTo后面的括号），而是把它赋给常量changeName。

changeName是什么呢？想知道答案，只要按住Option键再点击changeName即可。你会看到类似于图15-12的内容。（playground的运行结果侧边栏里也有同样的函数签名。）

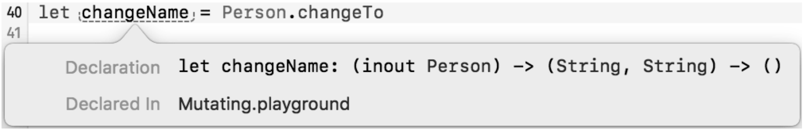


图15-12　mutating方法的签名

这个签名有什么意义？简单地说，它告诉你changeName是接受一个参数并返回一个函数的函数。具体点说，changeName持有的函数的唯一参数是结构体Person的一个实例，以inout参数的形式传入。这个函数返回的函数接受两个参数：表示新姓氏的字符串和表示新名字的字符串。返回的函数没有返回值。

回忆一下第12章的inout参数，它能让函数修改传递进来的参数。在函数内对inout参数所做的改动会在调用结束后保留下来。换句话说，改动会取代参数的原始值。

把所有的信息整合起来，结论就是：mutating方法的第一个参数是self，并以inout的形式传入。因为值类型在传递的时候会被复制，所以对于非mutating方法，self其实是值的副本。为了进行修改，self需要被声明为inout，而mutating就是Swift编译器让你完成这个任务的工具。

输入代码清单15-24所示的代码可以说明这一点，也可以看一下changeName的实际应用。

代码清单15-24　changeName的实际应用

```
...
let friendlyGreetingFor = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(mattGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeTo(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

var p = Person()
let changeName = Person.changeTo
let changeNameFromMattTo = changeName(p)
changeNameFromMattTo("John", "Gallagher")
p.firstName // "John"
```

首先调用changeName函数，传入要修改的Person实例。结果就是对Person实例调用changeName里的函数然后赋给changeNameFromMattTo。记住，inout参数要加上前缀&，确保传给函数的是实例的引用。接着，调用changeNameFromMattTo(_:_:)并给两个参数传入两个字

符串（"John"和"Gallagher"），分别是名字和姓氏。最后，打印函数调用的结果，确认p的名字改成了John。

本节的主要内容是说明关键字mutating做了什么。在实际编程中，不太可能会用这么麻烦的方法修改结构体。如代码清单15-25所示，删除新代码，直接用改名方法。结果是一样的。

代码清单15-25　回到实例方法

```
...
let friendlyGreeting = greeting("Hello,")
let mattGreeting = friendlyGreetingFor("Matt")
print(newGreeting)

struct Person {
    var firstName = "Matt"
    var lastName = "Mathias"

    mutating func changeTo(firstName: String, lastName: String) {
        self.firstName = firstName
        self.lastName = lastName
    }
}

var p = Person()
let changName = Person.changeTo
let changeNameFromMattTo = changeName(&p)
changeNameFromMattTo("John", "Gallagher")
p.changeTo(firstName: "John", lastName: "Gallagher")
p.firstName // "John"
```

第16章 属　　性

第15章简单提到了属性。尽管第15章的重点是结构体和类，但是我们也在类型上添加了基本的存储属性用以表示数据。本章会详细讨论属性，并加深你对如何在自定义类型中使用属性的理解。

属性能够把值关联到类型上，从而模拟类型所表示的实体的性质。属性的值可以是常量，也可以是变量。类、结构体和枚举都可以有属性。

属性分为两种：*存储*（stored）属性和*计算*（computed）属性。存储属性可以有默认值，计算属性则根据已有信息返回某种计算结果。我们可以观察属性的变化，并在属性被赋予新值时执行特定的代码。我们甚至可以设立规则，用来决定属性对应用中其他文件的可见度。

简单地说，属性功能强大而又灵活。我们来看看属性能做什么。

16.1　基本的存储属性

存储属性是属性的最基本形式。要了解存储属性的用法，我们需要把第15章写的类型细细剖析一番。先复制一个MonsterTown工程。启动Xcode并打开复制的工程。

打开Town.swift，看一下`population`属性的声明：`var population = 5_422`。这行代码有三个重要的细节。

- `var`表示这个属性是变量。
- `population`的默认值是`5_422`。
- `population`是存储属性，其值可以被读写。

怎么判断`population`是存储属性呢？因为它持有信息——镇子的人口。这就是存储属性的作用：存储数据。

`population`是*读写*属性。你既可以从属性中读取值，也可以给属性设置值。存储属性也可以是*只读*的，其值不可变。只读属性有一个我们更熟悉的名字：常量。

用`let`创建一个只读属性，存储镇子所在的区域信息（如代码清单16-1所示）。镇子毕竟不能移动，总是在同一区域。

代码清单16-1　添加区域常量（Town.swift）

```
struct Town {
    let region = "South"
```

```
        var population = 5_422
        var numberOfStoplights = 4

        func printDescription() {
            print("Population: \(population);
                   number of stoplights: \(numberOfStoplights)")
        }

        mutating func changePopulation(by amount: Int) {
            population += amount
        }
    }
```

region的实现现在没有问题，不过有点瑕疵。本章后面会解释这个问题并提供一个更好的解决方案。

16.2 嵌套类型

嵌套类型（nested type）是定义在另一个类型内部的类型，常用来支持某种类型的功能，而且不会单独在那种类型外面使用。你已经见到过嵌套函数，嵌套类型与其类似。

枚举通常是嵌套的。在Town.swift中，创建新的枚举Size。这个枚举可以用来和一个稍后添加的属性一起计算镇子的规模是小、中还是大。确保在Town结构体内定义枚举，如代码清单16-2所示。

代码清单16-2 创建Size枚举（Town.swift）

```
struct Town {
    let region = "South"
    var population = 5_422
    var numberOfStoplights = 4

    enum Size {
        case small
        case medium
        case large
    }

    func printDescription() {
        print("Population: \(population);
               number of stoplights: \(numberOfStoplights)")
    }

    mutating func changePopulation(by amount: Int) {
        population += amount
    }
}
```

Size决定Town的规模。在这个嵌套类型可用之前，Town的实例需要初始化population。目前我们见过的所有属性都是在实例创建时就计算好了值。下一节会介绍一种新的属性，这种属性会延迟计算，直到所需的信息准备好。

16.3　惰性存储属性

有时候我们不能马上给存储属性赋值。或许所需的信息已经有了，但是立即计算属性的值会消耗很多的内存或时间；或许属性会依赖类型外部的因素，只有实例创建后这个因素才可用。我们称这种情况为**惰性加载**（lazy loading）。

对于属性来说，惰性加载意味着属性的值只在第一次访问的时候才会出现。这种延迟把属性的值的计算推迟到实例初始化后。这意味着`lazy`属性必须声明为`var`，因为它们的值会发生变化。

创建`lazy`属性`townSize`，类型是`Size`，因为其值会是枚举`Size`的实例（如代码清单16-3所示）。再次确认新的属性是定义在`Town`类型内部的。

代码清单16-3　创建`townSize`（Town.swift）

```
struct Town {
    ...
    enum Size {
        case small
        case medium
        case large
    }
    lazy var townSize: Size = {
        switch self.population {
        case 0...10_000:
            return Size.small

        case 10_001...100_000:
            return Size.medium

        default:
            return Size.large
        }
    }()

    func printDescription() {
        print("Population: \(population);
               number of stoplights: \(numberOfStoplights)")
    }
    ...
}
```

`townSize`看起来和之前的属性不太一样。我们来一步一步进行分析。

首先，我们把`townSize`标记为`lazy`。这意味着程序只有在第一次访问`townSize`的时候才会计算它的值。稍后会详细解释原因。

接着，声明属性的类型是`Size`。我们没有像以往一样直接设置属性的值。比如，我们没有写成`myTown.townSize = Size.small`，而是利用嵌套枚举`Size`和闭包来计算给定人口数的镇子规模。

我们利用闭包的结果为`townSize`设置了一个默认值，就是镇子的规模（注意左花括号：`lazy var townSize: Size = {`）。回忆一下，函数和闭包是一等公民类型，而属性可以引用函数和闭包。

这里能用闭包是因为镇子的规模依赖于其人口数量。这个闭包用了switch语句判断规模。换句话说，闭包会根据实例的population（self.population）判断规模。这一行的self很重要，我们稍后还会讲到。

在switch语句内部有三个分支。人口数为0~10 000的是小镇，10 001~100 000是中型镇。第三个是默认分支，我们认为人口数大于100 000的镇子都是大型镇。分支体返回与给定人口匹配的Size实例。

注意townSize的闭包在右花括号后面用空的圆括号结尾。这对圆括号和惰性加载的标记一起确保了Swift会在我们第一次访问这个属性的时候调用闭包并将结果赋给它。如果省略了圆括号，那就只是把闭包赋给townSize属性。有了圆括号，闭包会在我们第一次访问townSize属性的时候得到执行。

最后，我们回过头来解释self.population中self的重要性，以及townSize为什么必须是惰性的。这个闭包必须引用self才能在闭包内访问到这个实例的population属性。为了让闭包能安全地访问self，编译器必须知道self已经初始化完成了（第17章会详细介绍）。把townSize标记为lazy是告诉编译器这个属性不是创建self所必需的；如果它不存在，就应该在它第一次被访问的时候创建。这就告诉编译器：当闭包被调用时，self肯定已经可用了。

切换到main.swift，练习使用lazy属性，如代码清单16-4所示。

代码清单16-4　使用惰性townSize属性（main.swift）

```
var myTown = Town()
let myTownSize = myTown.townSize
print(myTownSize)
myTown.changePopulation(by: 500)
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printDescription()
```

这里创建了常量myTownSize来保存myTown的规模信息。这行代码会访问惰性属性townSize，造成闭包运行。在闭包根据myTown的population得到结果后，就会把枚举Size的实例赋给myTownSize。然后把常量myTownSize的值打印出来。结果是，程序运行后会打印small到控制台。

一定要注意，标记为lazy的属性只会被计算一次。lazy的这个特性意味着，即使改变myTown的population，也不会造成townSize重新计算。要验证这一点，把myTown的population增加1 000 000，然后打印并查看myTown的规模（如代码清单16-5所示）。

代码清单16-5　修改myTown的population不会造成townSize变化（main.swift）

```
var myTown = Town()
let myTownSize = myTown.townSize
print(myTownSize)
myTown.changePopulation(by: 500)(by: 1_000_000)
print("Size: \(myTown.townSize); population: \(myTown.population)")
```

```
let fredTheZombie = Zombie()
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printDescription()
```

运行程序，你会看到控制台有这么一行输出：Size: small; population: 1005422。尽管人口显著增加了，myTown的规模仍然没有变化。这个差异是townSize的lazy特性造成的。属性只会在第一次被访问的时候计算，后面不会再重新计算。

myTown中这种population和townSize之间的差异是不受欢迎的。如果lazy意味着myTown不会让townSize反映population的变化，似乎不应该把townSize标记为lazy。

在合适的场景下，惰性加载是强大的工具；在第27章还会用到它。不过在本例中，这不是最好的工具，*计算属性*是更好的选择。

16.4　计算属性

任何自定义的类、结构体和枚举都可以使用计算属性。计算属性不会像之前的属性那样存储值，而是提供一个*读取方法*（getter）来获取属性的值，并可选地提供一个*写入方法*（setter）设置属性的值。这个区别能让计算属性的值发生变化，不像惰性存储属性的值那样无法变化。

用只读的计算属性替换Town类型的属性townSize的定义，如代码清单16-6所示。与只读的存储属性不同，只读的计算属性是用var定义的。

代码清单16-6　使用计算属性（Town.swift）

```
...
~~lazy~~ var townSize: Size ~~=~~ {
    get {
        switch self.population {
        case 0...10_000:
            return Size.small

        case 10_001...100_000:
            return Size.medium

        default:
            return Size.large
        }
    }
}~~()~~
...
```

这里的变化看起来很小。我们删除了第一行的lazy和=，在第二行添加了get语句（还有switch语句后面的右括号），还删除了最后一行的括号。只有这么多。但是这点小改动的影响很大。

townSize现在定义为计算属性，跟所有的计算属性一样，用var关键字声明。它提供了一个默认读取方法，用的是之前的switch语句。注意，需要显式地声明计算属性的类型是Size。计

算属性必须有类型信息，这样编译器才能知道属性的读取方法返回什么信息。

访问这个属性要用点语法（myTown.townSize），所以main.swift中的代码不用修改。访问这个属性会执行townSize的读取方法，由此可以利用myTown的population计算townSize。再次运行程序，你会在控制台看到：Size: large; population: 1005422。

townSize现在是只读的计算属性。换句话说，我们不能直接设置townSize，它只能根据读取方法中定义的计算过程获取并返回一个值。只读属性非常适合这种情形，因为我们希望myTown能根据实例的人口数计算townSize，而人口数可能在运行时发生变化。

读取方法和写入方法

计算属性也可以声明拥有读取方法和写入方法。读取方法能让你从属性读取数据，写入方法能让你向属性写入数据。同时拥有读取方法和写入方法的属性被称为读写属性。打开Monster.swift，给Monster添加一个计算属性，如代码清单16-7所示。

代码清单16-7 创建拥有读取方法和写入方法的victimPool计算属性（Monster.swift）

```
class Monster {
    var town: Town?
    var name = "Monster"
    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }

    func terrorizeTown() {
        if town != nil {
            print("\(name) is terrorizing a town!")
        } else {
            print("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

假设现在要让每个Monster的实例记录潜在的受害者群体。这个数字应该等于受怪兽侵扰的镇子人口数。相应地，victimPool应该是拥有读取方法和写入方法的计算属性。跟之前一样，用var声明计算属性，给出具体的类型信息。在本例中，victimPool是Int。

在属性的定义中，和townSize一样用get来定义读取方法。读取方法利用nil合并操作符检查Monster实例是否有正在侵扰的镇子，有的话就返回镇子的人口，没有就返回0。

计算属性的写入方法写在set块里。注意新语法：set(newVictimPool)。在圆括号中指定newVictimPool意味着我们提供了一个有明确名字的新变量，在写入方法的实现中可以引用这个变量。比如，用可空链式调用确保Monster的实例有一个镇子，然后设置镇子的人口使其和newVictimPool保持一致。如果你没有明确地给变量命名，那么Swift提供变量newValue来持有

相同的信息。

切换回main.swift来使用新的计算属性。在文件末尾添加如代码清单16-8所示的代码。

代码清单16-8　使用victimPool（main.swift）

```
...
print("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
print("Victim pool: \(fredTheZombie.victimPool);
       population: \(fredTheZombie.town?.population)")
```

第一行代码练习使用计算属性的读取方法。运行程序，终端会打印`Victim pool: 1005412`。第二行（`redTheZombie.victimPool = 500`）用写入方法改变`fredTheZombie`的`victimPool`。最后，再用读取方法把`victimPool`打印到终端。在控制台中，`victimPool`应该更新到了500，镇上的人口数也应该与之相符。

注意镇子人口的输出是`Optional(500)`，这跟`victimPool`的输出不一样，因为`fredTheZombie`的`town`可空。如果对这一点感到好奇，第22章会系统讲解可空类型。

16.5　属性观察者

Swift提供了一个有意思的特性，叫作*属性观察*（property observation）。属性观察者会观察并响应给定属性的变化。属性观察对于任何自定义的存储属性和任何继承的属性都可用。自定义的计算属性不能用属性观察。（但是我们对这类计算属性的读取方法和写入方法的定义有完全的控制权，所以可以用它来响应变化。）

想象处于困境中的镇民无法入眠，他们希望镇长采取措施保护自己免受这场灾害的侵袭。镇长的第一个举措是追踪针对镇民的袭击。属性观察者非常适合这种任务。

用以下两种方式可以观察属性的变化：

- 通过`willSet`观察属性即将发生的变化；
- 通过`didSet`观察属性已经发生的变化。

为了记录镇子正在遭受的袭击数量，镇长决定密切关注人口的变化。在Town.swift中用`didSet`观察者可以在属性得到新值时马上得到通知，如代码清单16-9所示。

代码清单16-9　观察人口变化（Town.swift）

```
struct Town {
...
    var population = 5_422 {
        didSet(oldPopulation) {
            print("The population has changed to \(population)
                   from \(oldPopulation).")
        }
    }
    ...
}
```

属性观察者的语法有点像计算属性的读取方法和写入方法。响应变化的代码放在花括号中。上例为旧人口数创建了一个参数名：oldPopulation。didSet观察者能让我们访问属性的旧值。如果不指定新名字，Swift会自动把参数命名为oldValue。（对于willSet观察者来说，Swift会生成newValue参数。）

这个属性观察者会在镇子人口数每次发生变化的时候将其打印到控制台。这意味着在fredTheZombie袭击镇子后我们会看到人口变化日志。运行程序并观察控制台，其输出应该与下面类似，每次population变化都会产生日志。

```
small
The population has changed to 1005422 from 5422.
Size: large; population: 1005422
The population has changed to 1005412 from 1005422.
Monster is terrorizing a town!
Population: 1005412; number of stoplights: 4
Victim pool: 1005412
The population has changed to 500 from 1005412.
Victim pool: 500; population: Optional(500)
Program ended with exit code: 0
```

由于使用属性观察者来打印人口数的变化，我们不再需要在main.swift中更新完victimPool后打印人口数变化了。把main.swift结尾处调用print()的相应代码删掉，如代码清单16-10所示。

代码清单16-10 删除print()中的population（main.swift）

```
...
print("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
print("Victim pool: \(fredTheZombie.victimPool);
        population: \(fredTheZombie.town?.population)")
```

16.6 类型属性

到目前为止，我们接触的都是实例属性。只要创建一个类型的实例，该实例就会分配到不同于同类型其他实例的属性。实例属性适合存储和计算一个类型的实例的值，但是属于类型本身的值怎么办呢？

可以定义*类型属性*（type property）。这类属性对类型是通用的——它们的值在同类型实例间共享。这类属性适合存储对于所有实例来说都相同的信息。比如，Square类型的所有实例都有四条边，所以Square的边数可以存储在类型实例中。

值类型（结构体和枚举）既可以有存储类型属性，也可以有计算类型属性。跟类型方法一样，值类型的类型属性以关键字static开头。

回忆本章前面在Town类型上创建的表示镇子区域的只读属性。使用常量实例属性使得Town的每个实例都位于同一区域：南方。region是类型属性会更好，因为我们用它来表示对这个类型通用的概念“南方”。修改Town，如代码清单16-11所示。

代码清单16-11 把region改为存储类型属性（Town.swift）

```
struct Town {
    static let region = "South"
    ...
}
```

存储类型属性必须有默认值。这个要求是合理的，因为类型没有初始化方法（第17章会讨论初始化方法）。这意味着存储类型属性给任何调用者提供值之前必须准备好必要的信息。在这里，`region`被赋值为`South`。

类也可以有存储类型属性和计算类型属性，语法跟结构体一样用`static`。子类不能覆盖父类的类型属性。如果希望子类能为某个属性提供自己的实现，那就用`class`关键字。

在15.10节，我们展示了`Zombie`类型可以发出恐怖声音的类型方法，如下所示。

```
class Zombie: Monster {
    class func makeSpookyNoise() -> String {
        return "Brains..."
    }
```

注意`makeSpookyNoise()`没有参数，这让它非常适合改写成计算类型属性。打开Zombie.swift，增加计算类型属性返回僵尸流行语，如代码清单16-12所示。

代码清单16-12 创建spookyNoise计算类型属性（Zombie.swift）

```
class Zombie: Monster {
    class var spookyNoise: String {
        return "Brains..."
    }
    var walksWithLimp = true
    ...
}
```

类型级别计算属性的定义跟类型方法很像，主要的区别在于用关键字`var`而不是`func`，以及没有圆括号。

上面的代码还有一个新看点，那就是用了读取方法的快捷语法。如果计算属性没有写入方法，就可以省略计算属性定义中的`get`，并直接返回所需的计算值。

切换到main.swift，在文件末尾增加一行代码打印`Zombie`类型的`spookyNoise`属性，如代码清单16-13所示。

代码清单16-13 “Brains...”（main.swift）

```
...
print("Victim pool: \(fredTheZombie.victimPool)")
fredTheZombie.victimPool = 500
print("Victim pool: \(fredTheZombie.victimPool)")
print(Zombie.spookyNoise)
```

运行程序。好恐怖。

为了说明`class`类型属性能被子类覆盖，给`Monster`添加`spookyNoise`计算类型属性，如代

码清单16-14所示。

代码清单16-14　通用的Monster声音（Monster.swift）

```
class Monster {
    class var spookyNoise: String {
        return "Grrr"
    }
    var town: Town?
    var name = "Monster"
    ...
}
```

切换回Zombie.swift，你会发现编译器报错了。如果点击编辑器区域左边的红色感叹号，错误信息会显示出来（如图16-1所示）。



图16-1　覆盖错误

现在，Zombie覆盖了父类的计算类型属性。因为这个类型属性用了class关键字，所以子类提供自己的spookyNoise定义是完全没问题的，只不过要给Zombie的spookyNoise定义添加override关键字。

如代码清单16-15所示修改之后，编译器报错就消失了。

代码清单16-15　覆盖spookyNoise（Zombie.swift）

```
class Zombie: Monster {
    override class var spookyNoise: String {
        return "Brains..."
    }
    var walksWithLimp = true
    ...
}
```

构建并运行程序，结果应该跟之前完全一样。

前面我们提到过类可以有类型层面的静态属性。这种属性跟类型层面的类属性不太一样。

所有怪物的一个决定性特征是可怕。给Monster类添加静态属性表示这一事实，如代码清单16-16所示。

代码清单16-16　所有的Monster都可怕（Monster.swift）

```
class Monster {
    static let isTerrifying = true
    class var spookyNoise: String {
        return "Grrr"
    }
    ...
}
```

我们给Monster添加了新的静态属性，表示所有怪物的本质特征就是可怕。因为这个属性加到了Zombie的父类Monster上，所以Zombie也能用。给main.swift添加如代码清单16-17所示代码来看一下实际使用。

代码清单16-17　逃离Zombie（main.swift）

```
...
print(Zombie.spookyNoise)
if Zombie.isTerrifying {
    print("Run away!")
}
```

正如你所见，可以用点语法访问Zombie的isTerrifying属性。如果Zombie吓人，那就逃远点。

构建并运行程序，控制台警告你快逃（Run away!）。

静态属性和类型属性最大的区别是静态属性无法被子类覆盖。把这个类型属性声明为静态常量是很明显的：Monster都可怕，而子类无法改变这一点。

16.7　访问控制

有时候，我们希望程序的某部分代码对其他部分不可见。事实上，对代码访问有细粒度的控制是常见需求。我们可以给某些组件访问其他组件的一定级别的访问权限，这被称为访问控制（access control）。

举个例子，你想隐藏或暴露一个类的方法。假设现在有个属性只在类定义的内部使用。如果有一个外部类型错误修改了这个属性，就可能会出问题。有了访问控制，你可以管理那个属性的可见度，使其对程序的其他部分不可见。这样会把属性的数据封装好，避免外部代码插手。

访问控制是围绕着两个重要且相关的概念组织的：模块（module）和源代码文件（source file）。对工程的文件和组织来说，这是应用的核心构件。

模块是分发代码的单位。你应该能回想起来playground开头有import Cocoa，而Swift文件中则有import Foundation。这些是框架，作用是把一组执行相关任务的类型打包在一起。比如，Cocoa就是用来开发macOS应用的框架。用Swift的import关键字可以把一个模块引入另一个模块，如上例所示。

另一方面，源代码文件是更独立的单元。源代码文件表示一个文件，并且存在于特定的模块中。把单个类型放进一个源代码文件是个好习惯。这不是强制性的，但是做有助于组织工程。

Swift提供五个访问层级（如表16-1所示）。

表16-1　Swift的访问控制

访问层级	描　　述	对……可见	能在……中继承
open	实体对模块内的所有文件以及引入了该模块的文件都可见，并且可以继承	实体所在的模块以及引入实体所在模块的模块	实体所在的模块以及引入实体所在模块的模块

（续）

访问层级	描　　述	对……可见	能在……中继承
Public	实体对模块内的所有文件以及引入了该模块的文件都可见	实体所在的模块以及引入实体所在模块的模块	实体所在的模块
internal（默认）	实体对模块内的文件可见	实体所在的模块	实体所在的模块
fileprivate	实体只对所在的源文件可见	实体所在的文件	实体所在的文件
private	实体只对所在的作用域可见	所在的作用域	所在的作用域

open访问层级限制最少，而private访问层级限制最多。通常来说，一个类型的访问层级必须与其属性和方法的访问层级一致。属性的访问层级不能比其所在的类型限制更少。比如说，一个访问层级是internal的属性不能在一个private访问层级的类型中声明。与之类似，函数的访问控制也不能比其参数列表限制更少。如果破坏了这些条件，编译器会报错。

Swift指定internal是默认访问层级。有了默认访问层级，就不需要为每个类型、属性和方法都明确声明访问控制了。把internal作为默认层级是合理的，因为一般用Swift写Cocoa和iOS应用，两者的源代码一般都用单个模块。因此，只有在需要比internal更强或更弱的访问控制时才需要指定。

来看一下private层级的实际使用。在Zombie上创建Bool属性isFallingApart，给其赋默认值false。这个属性会记录实例的身体完整性（毕竟，僵尸的一些身体部位有时候会掉下来）。这个属性实际上不需要暴露给程序的其他部分，因为这是Zombie类的实现细节。因此，把它设置为private，如代码清单16-18所示。

代码清单16-18　散架这件事是私有的（Zombie.swift）

```
class Zombie: Monster {
    override class var spookyNoise: String {
        return "Brains..."
    }
    var walksWithLimp = true
    private var isFallingApart = false

    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(by: -10)
        }
        super.terrorizeTown()
    }
}
```

创建属性后就可以在terrorizeTown()方法中使用了。我们检查isFallingApart是否为假；如果为假，那么这个实例可以侵扰镇子，否则不能。

isFallingApart在terrorizeTown()中可见，因为这个属性被声明为private，也就是说任何同一作用域内定义的实体都能访问isFallingApart。因为isFallingApart是Zombie的一个属性，所以在同一层级定义的任何属性或方法都能访问这个新属性。不过，在Zombie类外无

法访问isFallingApart。这个属性是该类的私有实现。

控制读取方法和写入方法的可见度

如果属性既有读取方法又有写入方法，可以分别控制它们的可见度；不过在默认情况下，读取方法和写入方法的可见度相同。这里，isFallingApart的读取方法和写入方法都是私有的。

不过，你可能想让工程中的其他文件能够判断Zombie是否要散架了，而不想给它们改变是否散架状态的权限。把isFallingApart改为internal的读取方法和private的写入方法，如代码清单16-19所示。

代码清单16-19　把读取方法设置为internal，把写入方法设置为private（Zombie.swift）

```
class Zombie: Monster {
    ...
    private internal private(set) var isFallingApart = false
    ...
}
```

用internal private(set)语法可以指定读取方法是内部访问层级，写入方法是私有访问层级。两个关键字可以是public、internal或private，只要写入方法的可见度不比读取方法更高就可以。举个例子，如果读取方法是internal，那就不能写public(set)，因为公开访问层级的可见度比内部访问层级更高。此外，Zombie默认是internal，因为我们没有为其指定访问层级。这意味着给isFallingApart的读取方法和写入方法设置public的话，编译器就会警告你其所在的类的可见度是internal。

这段代码可以稍微简化一下。如果读取方法不写修饰关键字，那么其访问控制就默认是internal，正是我们想要的。重构Zombie，使读取方法使用默认可见度（internal），写入方法使用私有可见度（如代码清单16-20所示）。

代码清单16-20　使用默认的读取方法可见度（Zombie.swift）

```
class Zombie: Monster {
    ...
    internal private(set) var isFallingApart = false
    ...
}
```

使用默认值不会有任何变化，只是节省了打字量。isFallingApart的读取方法仍然对工程中其他文件可见，而写入方法仍然只对Zombie.swift内部可见。

本章介绍了大量内容，花些时间吸收一下。你学到的东西包括：

- 属性语法
- 存储属性和计算属性
- 只读属性和读写属性
- 惰性加载和惰性属性
- 属性观察者

- 类型属性
- 访问控制

属性是Swift编程的核心概念。熟悉这些内容很有帮助，下面的挑战练习可以帮助你掌握这些重要概念。

16.8　青铜挑战练习

镇长很忙，不必关注每次出生和搬迁。毕竟镇子处于危机中，只有当人口数减少时才打印人口变化。

16.9　白银挑战练习

新建一个叫作`Mayor`的结构体类型。`Town`类型应该有一个属性叫`mayor`，持有一个`Mayor`类型的实例。

每次人口变化就通知`mayor`。如果镇人口减少，就让`Mayor`实例打印如下信息到控制台：`I'm deeply saddened to hear about this latest tragedy. I promise that my office is looking into the nature of this rash of violence.`。如果人口增加，镇长什么都不用做。

（提示：你需要给`Mayor`类型定义一个新的实例方法来完成这个练习。）

16.10　黄金挑战练习

镇长也是人，`Mayor`自然会在因`Zombie`袭击而造成镇子人口损失的时候感到紧张。给`Mayor`类型创建一个实例属性`anxietyLevel`，其类型应该是`Int`，默认值是0。

每次`Mayor`得到有`Zombie`袭击的通知时就增加`anxietyLevel`属性。最后，作为镇长，他肯定不希望自己的焦虑被旁人看出来，所以把这个属性标记为`private`。确认main.swift无法访问这个属性。

第 17 章 初　始　化

17

初始化是设置类型实例的操作，包括给每个存储属性初始值，以及一些其他准备工作。完成这个过程后，实例就可以使用了。

到目前为止，我们创建类型的方式都差不多。属性要么有默认值，要么是按需计算。我们没有自定义过初始化方法，也没有专门学习过。

控制类型实例的创建过程是常见的需求。比如，如果能让实例的属性立即得到正确的值，那无疑是再好不过了。之前我们都是给实例的存储属性一个默认值，并在创建实例后修改。这种做法不够优雅，*初始化方法*（initializer）能帮助我们在创建实例的同时为其赋予合适的值。

17.1　初始化方法语法

结构体和类的存储属性在初始化完成的时候需要有初始值。这个要求能解释为什么我们之前要给所有的存储属性设置默认值。如果不这么做，编译器会报错，告诉你类型的属性还不可用。为类型定义初始化方法是确保实例创建时其属性有值的另一个方法。

初始化方法的语法和我们前面接触过的方法不大一样。初始化方法用关键字`init`表示。虽然初始化方法是类型的方法，但是前面没有`func`关键字。初始化方法的语法如下所示：

```
struct CustomType {
    init(someValue: SomeType) {
        // 初始化代码
    }
}
```

结构体、枚举和类都使用这种通用语法，没什么区别。在上例中，初始化方法有一个参数`someValue`，参数类型是`SomeType`。初始化方法通常有一个或多个参数，不过也可以没有参数（这种情况下，`init`关键字后面是空的圆括号）。

初始化方法的实现在花括号中定义，这与本书中的普通函数和方法一样。不过跟方法不同的是，初始化方法没有返回值。初始化方法的任务是给类型的存储属性赋值。

17.2　结构体初始化

结构体既可以有默认初始化方法，也可以有自定义初始化方法。在使用结构体时，通常可以

利用默认初始化方法，不过有些场景需要自定义初始化过程。

17.2.1　结构体的默认初始化方法

还记得之前如何创建Town的实例吗？我们给属性设置了默认值；但是你不知道的是，其实我们利用了Swift编译器提供的*空初始化方法*（empty initializer，没有参数的初始化方法）。当你输入代码var myTown = Town()时，就用到了空初始化方法并为新实例的属性指定了默认值。

默认初始化方法还有一种形式，称作*成员初始化方法*（memberwise initializer）。对于类型的每个存储属性，成员初始化方法都有相应的参数。在这种情况下，不需要让编译器给新实例的属性设置默认值，而是利用自带的成员初始化方法为所有需要值的属性提供参数。

记住，初始化方法的一个基本目的是给类型的所有存储属性赋值，以便新实例可用。编译器会强制要求新实例的存储属性有值。如果没有为自定义结构体提供初始化方法，就必须通过默认值或成员初始化方法提供必要的值。

复制上一章的MonsterTown工程，这样就有现成的代码了。打开复制的工程并切换到main.swift。

在main.swift中，把Town的空初始化方法替换为成员初始化方法，再增加一个printDescription()调用，如代码清单17-1所示。不过先别着急运行程序去看控制台打印了什么。

代码清单17-1　使用成员初始化方法（main.swift）

```
...
var myTown = Town~~()~~(population: 10_000, numberOfStoplights: 6)
myTown.printDescription()
...
```

接下来查看Town.swift文件。注意，population和numberOfStoplights属性有默认值。这些默认值和我们为Town的成员初始化方法提供的参数值不同。

现在运行程序。输出跟你的预期一致吗？控制台中myTown的描述是这样的：Population: 10_000; number of stoplights: 6。这不是我们给Town存储属性提供的默认值。这些属性值是怎么从默认值变过来的？

myTown实例是用成员初始化方法创建的。Town的存储属性出现在初始化方法的参数列表中，这样就能为属性指定新值了。正如控制台显示的，你给初始化方法提供的值取代了默认值。

注意，Town的属性名在调用初始化方法时被用作外部参数名。Swift自动为每个初始化方法的每个参数提供外部参数名。这个约定很重要，因为Swift的初始化方法名字都一样：init。因此，函数名无法用来表示要调用的是哪个特定的初始化方法。参数名及其类型能帮助编译器区分不同的初始化方法，以便知道该调用哪个。

结构体的默认成员初始化方法很有用，因为这是Swift自动提供的，不需要写代码实现。结构体的这个优势让它们变得极具吸引力。不管怎么样，很多时候你也需要自定义类型的初始化方法。下面就来讲讲自定义初始化方法。

17.2.2　结构体的自定义初始化方法

现在是时候为Town写一个自定义初始化方法了。自定义初始化方法很强大，而能力越大，责任就越大。一旦写了自己的初始化方法，Swift就不会提供默认的初始化方法了。（和默认的成员初始化方法说再见吧！）你得负责让实例的属性有合适的值。

首先清扫一下代码，把所有属性的默认值都去掉。在介绍初始化方法之前，默认值很有用，因为它们能确保创建实例时属性有值。不过，现在它们就没那么有意义了。另外，把region改回实例属性——怪物大量出现，已经蔓延到南方以外了。最后，把镇子的region添加到printDescription打印的日志中，因为镇子的region现在可能随实例而不同。

打开Town.swift做上面这些改动，如代码清单17-2所示。

代码清单17-2　清扫代码（Town.swift）

```
struct Town {
    static let region = "South"

    var population = 5_422 {
        didSet(oldPopulation) {
            print("The population has changed to \(population)
                   from \(oldPopulation).")
        }
    }
    var numberOfStoplights = 4

    ...

    func printDescription() {
        print("Population: \(population); number of stoplights:
              \(numberOfStoplights); region: \(region)")
    }
...
```

删掉默认值后，你可能会注意到编译器在三个地方报错了，都是错误Type annotation missing in pattern。之前的代码利用了类型推断，配合默认值可以正常工作。一旦没有了默认值，编译器就不知道属性的类型信息了。我们需要明确指定类型，如代码清单17-3所示。

代码清单17-3　声明类型（Town.swift）

```
struct Town {
    let region: String

    var population: Int {
        didSet(oldPopulation) {
            print("The population has changed to \(population)
                   from \(oldPopulation).")
        }
    }
    var numberOfStoplights: Int
...
```

现在是时候创建自定义初始化方法了。稍后我们会从同一类型的另一个初始化方法中调用这

个初始化方法。现在先把代码清单17-4所示的初始化方法添加到Town。

代码清单17-4 添加成员初始化方法（Town.swift）

```
...
var numberOfStoplights: Int
init(region: String, population: Int, stoplights: Int) {
    self.region = region
    self.population = population
    numberOfStoplights = stoplights
}
enum Size {
    case small
    case medium
    case large
}
...
```

这里的初始化方法init(region:population:stoplights:)有三个参数，每个对应Town的一个存储属性。我们可以把参数中的值传递给属性。比如说，传递给region参数的值被赋给了region属性。因为初始化方法中的参数名和属性名一样，所以访问属性需要明确指定self。numberOfStoplights属性没有这个问题，所以只要把初始化方法的stoplights参数赋给numberOfStoplights属性就可以了。

注意，虽然region属性被声明为常量，但我们还是给它赋值了。Swift编译器允许在初始化过程中初始化常量属性。记住，初始化的目的是确保类型的属性在初始化完成之后有值。

到了这里，你可能已经注意到Xcode打开了左侧的问题导航器，告诉你发生了一个错误（如图17-1所示）。（如果问题导航器没有自动打开，点击导航区域的左起第四个图标可以打开。）你会看到错误发生在main.swift，跟编译器默认提供的初始化方法有关。

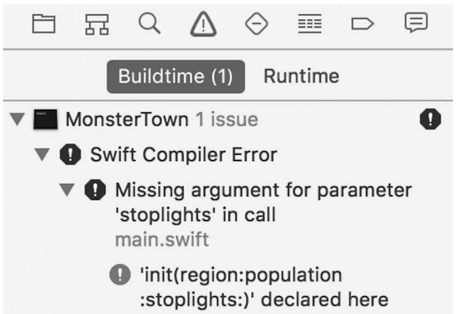


图17-1 在问题导航器中显示错误

切换回main.swift来修复这个问题。（点击顶部的文件夹图标可以回到工程导航器。）

之前编译器提供的成员初始化方法对numberOfStoplights属性对应的参数用了实际的名字。在Town的初始化方法中，我们把参数名简化为了stoplights。修改main.swift中的参数，还需要添加region参数，如代码清单17-5所示。

代码清单17-5　改正参数（main.swift）

```
...
var myTown = Town(region: "West",
                  population: 10_000,
                  numberOfStoplights stoplights: 6)
...
```

构建并运行程序，错误应该会消失，控制台输出的开头是`Population: 10_000; number of stoplights: 6; region: West`。

委托初始化

初始化方法的定义中可以包含对该类型其他初始化方法的调用。这个过程被称为*委托初始化*（initializer delegation），通常用来提供多种创建实例的路径。

对于值类型（也就是枚举和结构体）来说，委托初始化相对比较直观。因为值类型不支持继承，所以委托初始化只涉及调用所在类型的其他初始化方法。对于类来说，委托初始化则多少有些复杂，下面就来看一看。

切换到Town.swift，给这个类型添加一个新的初始化方法。这个方法会利用委托初始化，如代码清单17-6所示。

代码清单17-6　使用委托初始化（Town.swift）

```
...
init(region: String, population: Int, stoplights: Int) {
    self.region = region
    self.population = population
    numberOfStoplights = stoplights
}
init(population: Int, stoplights: Int) {
    self.init(region: "N/A", population: population, stoplights: stoplights)
}
enum Size {
    case small
    case medium
    case large
}
...
```

这里给`Town`定义了一个新的初始化方法。不过不同于之前的初始化方法，它只有两个参数：`population`和`stoplights`。

那么`region`属性呢？怎么为其设置值？

看一下新初始化方法的实现。在`self.init(region: "N/A", population: population, stoplights: stoplights)`这一行调用了`self`的另一个初始化方法。注意，我们传入了`population`和`stoplights`参数。因为没有`region`参数，所以我们得自己提供一个值。在本例中，通过指定字符串`"N/A"`表示没有区域信息传给初始化方法。

委托初始化可以避免代码重复。我们不需要重新输入把初始化方法的参数值传给类型属性的代码，只要调用另一个初始化方法就可以了。避免代码重复不仅能节省输入量，还可以减少bug。

当两个地方有相同代码的时候，只要有改动就必须记着同时修改两处。

我们说委托初始化为创建实例“定义了路径”。一个初始化方法调用另一个初始化方法，以提供创建实例所需的特定片段。最终，委托初始化会到达一个初始化方法，这里已经备齐了实例完全可用所需的东西。

因为有了自定义的成员初始化方法，编译器就不再默认提供了。这也不是什么坏事，有时候甚至是好事。举个例子，如果你要创建的镇子没有区域信息，那就可以用这个新的初始化方法。在本例中，可以用方便的新初始化方法通过`population`和`stoplights`参数为对应属性赋值，同时给`region`设置占位的值。

在main.swift中使用新的初始化方法，如代码清单17-7所示。

代码清单17-7　使用新初始化方法（main.swift）

```
...
var myTown = Town(region: "West", population: 10_000, stoplights: 6)
myTown.printDescription()
...
```

如果构建并运行程序，你会发现结果基本一样，只有一个差别：我们不再把`region`设置为特定的值，所以能从控制台看到其值是`N/A`。

17.3　类初始化

类初始化的通用语法看起来跟值类型差不多。不过，有必要注意一些不同的规则。存在这些额外的规则主要是因为类可以继承，而这必然会增加初始化的复杂度。

特别是类增加了*指定*（designated）初始化方法和*便捷*（convenience）初始化方法的概念。类的初始化方法一定是二者之一。指定初始化方法负责确保初始化完成前所有的属性都有值，以便实例可用。便捷初始化方法是指定初始化方法的补充，通过调用所在类的指定初始化方法来实现，主要作用通常是为某种特殊目的创建实例。

17.3.1　类的默认初始化方法

你已经见到过类的默认初始化方法。如果所有的属性都有默认值并且没有自定义初始化方法，类会得到一个默认的空初始化方法。与结构体不同，类没有默认的成员初始化方法。这解释了为什么之前要给类设置默认值：这样可以利用自带的空初始化方法。由此可以这样得到一个`Zombie`的实例：`let fredTheZombie = Zombie()`，其中的空圆括号表示这是一个默认初始化方法。

17.3.2　初始化和类继承

打开Monster.swift，修改这个类，为其添加一个初始化方法。还要删掉`name`属性的默认值`"Monster"`，如代码清单17-8所示。

代码清单17-8　初始化Monster（Monster.swift）

```
class Monster {
    ...
    var town: Town?
    var name = "Monster"
    var name: String
    var victimPool: Int {
        get {
            return town?.population ?? 0
        }
        set(newVictimPool) {
            town?.population = newVictimPool
        }
    }
    init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }
    func terrorizeTown() {
        if town != nil {
            print("\(name) is terrorizing a town!")
        } else {
            print("\(name) hasn't found a town to terrorize yet...")
        }
    }
}
```

这个初始化方法有两个参数：一个是可空的`Town`，另一个是怪物的名字。在初始化方法的实现中，这些参数的值被赋给类的属性。注意，因为参数`town`和属性名一样，所以设置属性值的时候还是要用`self`来访问。访问`name`不需要这么做，因为初始化方法的参数名和属性名不同。

添加完这个初始化方法后，你可能会注意到工具栏上显示有个编译错误（如图17-2所示）。点击红色图标，你会发现错误位于main.swift。切换到这个文件查看错误。



图17-2　工具栏上的错误

你会看到之前用`Zombie()`得到实例的形式已经无法通过编译器检查了。为什么呢？看一下错误信息：`Missing argument for parameter 'town' in call`。

错误显示编译器期望`Zombie`的初始化方法有`town`参数。这个期望可能有点奇怪，因为`Zombie`的初始化方法并不需要`town`。事实上，我们甚至还没有给这个类实现任何初始化方法，只是利用了编译器在所有属性都有默认值的情况下提供的空初始化方法。

这就是错误的原因：`Zombie`没有默认的空初始化方法了。为什么？因为*初始化方法自动继承*（automatic initializer inheritance）。

1. 初始化方法自动继承

一般来说，类不会继承父类的初始化方法。Swift的这个特性是希望避免子类在不经意间提供无法为所有属性赋值的初始化方法，因为子类经常会增加父类不存在的属性。让子类提供自己的

初始化方法可以避免实例被不完整的初始化方法初始化。

不过，类确实会在一些情况下自动继承父类的初始化方法。如果子类为所有新增的属性提供了默认值，那么在以下两种场景下，类会继承父类的初始化方法。

- 如果子类没有定义任何指定初始化方法，就会继承父类的指定初始化方法。
- 如果子类实现了父类的所有指定初始化方法（无论是通过显式实现还是隐式继承），就会继承父类的所有便捷初始化方法。

Zombie正好符合第一种场景。因为它为所有新增属性提供了默认值，又没有定义自己的指定初始化方法，所以继承了Monster唯一的指定初始化方法。而且因为Zombie继承了一个初始化方法，所以编译器不会提供默认的初始化方法，而我们之前用的就是默认的初始化方法。

Zombie继承的初始化方法的签名是init(town:monsterName:)，参数是town和monsterName。更新fredTheZombie的初始化方法，加上这两个参数就能消除编译错误。

代码清单17-9　更新fredTheZombie的初始化（main.swift）

```
...
let fredTheZombie = Zombie(town: myTown, monsterName: "Fred")
fredTheZombie.town = myTown
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printDescription()
...
```

从现在开始，要创建Monster或Zombie的实例，就需要给其town和name属性赋值了。构建并运行应用，错误应该已经消失了，结果跟之前是一样的。

2. 类的指定初始化方法

类的主要初始化方法是指定初始化方法。指定初始化方法的一部分作用是确保类属性在初始化完成前都有值。如果类有父类，那么子类的指定初始化方法必须调用父类的指定初始化方法。

Monster已经有指定初始化方法了：

```
init(town: Town?, monsterName: String) {
    self.town = town
    name = monsterName
}
```

指定初始化方法不需要修饰，也就是说不需要在init前面放置特殊的关键字。这样从语法上就可以区分指定初始化方法和便捷初始化方法，因为后者用关键字convenience表示。

Monster的初始化方法确保在初始化完成前所有的属性都有值。目前，Zombie把默认值给了它的所有属性（除了从Monster继承的）。因此，Monster的初始化方法对Zombie也适用。不过，最好还是给Zombie定义自己的初始化方法，以便定制初始化过程。

首先删除Zombie属性的默认值，如代码清单17-10所示。

代码清单17-10　删除默认值（Zombie.swift）

```
class Zombie: Monster {
    override class var spookyNoise: String {
        return "Brains..."
```

```
    }

    var walksWithLimp = true
    var walksWithLimp: Bool
    private(set) var isFallingApart = false
    private(set) var isFallingApart: Bool

    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
        super.terrorizeTown()
    }
    ...
}
```

删除默认值会导致编译错误：`Class 'Zombie' has no initializers`。在不设置默认值的情况下，Zombie需要一个初始化方法在初始化完成前为属性赋值。

给Zombie添加一个新的初始化方法来解决这个问题，如代码清单17-11所示。

代码清单17-11　为Zombie添加初始化方法（Zombie.swift）

```
class Zombie: Monster {
    override class var spookyNoise: String {
        return "Brains..."
    }

    var walksWithLimp: Bool
    private(set) var isFallingApart: Bool
    init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
        walksWithLimp = limp
        isFallingApart = fallingApart
        super.init(town: town, monsterName: monsterName)
    }
    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
        super.terrorizeTown()
    }
}
```

新的初始化方法会解决错误，因为现在能确保在初始化完成时Zombie的属性都有值。这里添加的代码分为两部分。首先，新的初始化方法通过`limp`和`fallingApart`参数为`walksWithLimp`和`isFallingApart`赋值。因为这些属性是Zombie特有的，所以指定初始化方法需要用合适的值将其初始化。

接着，调用Zombie父类的指定初始化方法。正如你在第15章看到的，`super`指向子类的父类。因此，语法`super.init(town: town, monsterName: monsterName)`会把Zombie的初始化方法参数`town`和`monsterName`的值传递给Monster的指定初始化方法。这样会调用Monster的初始化方法，确保Zombie的`town`和`name`属性被赋值。图17-3用图示说明了这种关系。

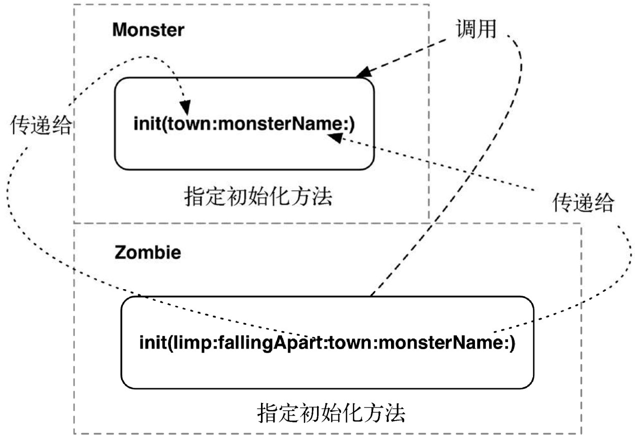


图17-3 调用super.init

你可能想知道为什么最后才调用父类的初始化方法。这是因为Zombie的初始化方法是指定初始化方法，它要负责为自己引入的所有属性赋值。给这些属性赋值后，子类的初始化方法要负责调用父类的初始化方法，以便父类初始化自己的属性。

main.swift有一个新的错误要处理。切换到这个文件，你会看到一个编译错误，告诉你Zombie的初始化方法缺少一个参数。更新fredTheZombie的初始化方法，加上Zombie的新增初始化方法的所有参数以修复错误，如代码清单17-12所示。

代码清单17-12 Fred走路是不是一瘸一拐的？他快散架了吗（main.swift）

```
...
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
...
```

现在fredTheZombie有足够信息初始化了，这些信息使得实例可用。

3. 类的便捷初始化方法

不同于指定初始化方法，便捷初始化方法不需要确保类的所有属性都有值，而是做完自己的工作后把信息传递给其他的便捷初始化方法或指定初始化方法。所有的便捷初始化方法都要调用所在类的其他初始化方法。最终，便捷初始化方法必须调用到指定初始化方法。一个类的便捷初始化方法和指定初始化方法会形成一条路径，类的存储属性通过这条路径收到初始值。

为Zombie添加便捷初始化方法，如代码清单17-13所示。这个初始化方法将提供参数，表示Zombie实例是否走路一瘸一拐以及是否快散架了。它还会省略town和monsterName参数：这个初始化方法的调用者只要负责提供参数所需的值就可以了。

代码清单17-13 使用便捷初始化方法（Zombie.swift）

```
...
init(limp: Bool, fallingApart: Bool, town: Town?, monsterName: String) {
```

```
        walksWithLimp = limp
        isFallingApart = fallingApart
        super.init(town: town, monsterName: monsterName)
    }
    convenience init(limp: Bool, fallingApart: Bool) {
        self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "Fred")
        if walksWithLimp {
            print("This zombie has a bad knee.")
        }
    }
    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
    }
    ...
```

用convenience关键字可以把初始化方法标记为便捷初始化方法。这个关键字告诉编译器：这个初始化方法需要把一部分工作委托给另一个初始化方法，直到调用到一个指定初始化方法。调用完成后，类的这个实例就可用了。

上例的便捷初始化方法调用了Zombie的指定初始化方法。它把接收到的参数limp和fallingApart的值传递过去。对于便捷初始化方法没有接收到的参数town和monsterName，就传递nil和"Fred"给Zombie的指定初始化方法。

便捷初始化方法调用了指定初始化方法之后，这个实例就完全可用了。因此我们可以检查walksWithLimp的值。如果试图在调用Zombie的指定初始化方法之前检查，编译器会报错：Use of 'self' in delegating initializer before self.init is called。这个错误告诉我们委托方初始化方法在self可用之前试图使用它来访问walksWithLimp。

图17-4显示了便捷初始化方法和指定初始化方法的关系。

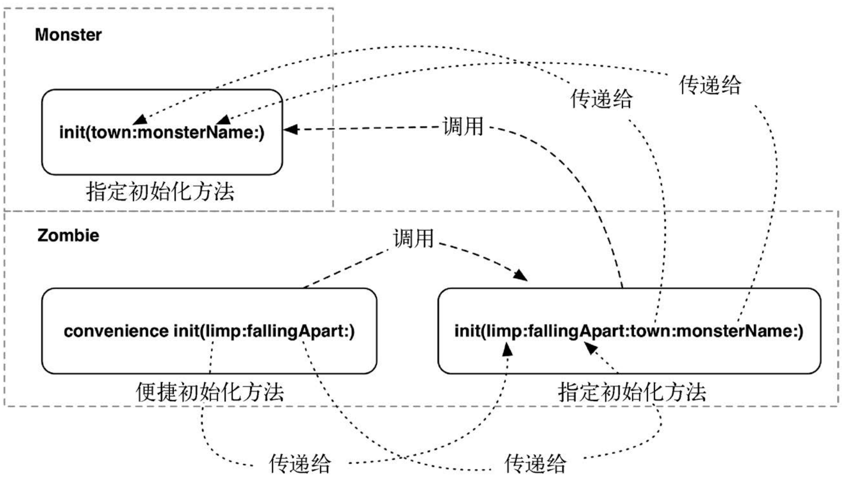


图17-4　委托初始化

现在可以用这个便捷初始化方法创建Zombie实例了。不过要记住，用这个便捷初始化方法所创建Zombie实例的town属性是nil，name属性是"Fred"。切换到main.swift并用它来创建实例，如代码清单17-14所示。

代码清单17-14　创建一个便捷的僵尸（main.swift）

```
...
let fredTheZombie = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombie.terrorizeTown()
fredTheZombie.town?.printDescription()

var convenientZombie = Zombie(limp: true, fallingApart: false)
...
```

构建并运行程序，你会看到convenientZombie的一个膝盖坏了。

17.3.3　类的必需初始化方法

一个类可以要求其子类提供特定的初始化方法。举个例子，假设你想让Monster的所有子类都提供怪物的名字和侵扰的镇子（如果怪物还没有找到镇子，那就是nil）。要做到这一点，只要用关键字required标记初始化方法即可，表示所有的子类都必须提供这个初始化方法。

切换到Monster.swift进行修改，如代码清单17-15所示。

代码清单17-15　把town和monsterName设为必需（Monster.swift）

```
class Monster {
    ...

    var victimPool: Int {
        ...
    }
    required init(town: Town?, monsterName: String) {
        self.town = town
        name = monsterName
    }

    func terrorizeTown() {
        ...
    }
}
```

Monster唯一的指定初始化方法现在是必需的，子类必须实现它。

不幸的是，Xcode的工具栏显示这个改动产生了一个编译错误。点击红色图标显示问题导航器，可以看到错误，错误信息是'required' initializer 'init(town:monsterName:)' must be provided by subclass of 'Monster'。它告诉你Zombie还没有实现新增的必需初始化方法。切换到Zombie.swift实现这个初始化方法，如代码清单17-16所示。

代码清单17-16　添加必需初始化方法（Zombie.swift）

```
...
convenience init(limp: Bool, fallingApart: Bool) {
```

```
        self.init(limp: limp, fallingApart: fallingApart, town: nil, monsterName: "Fred")
        if walksWithLimp {
            print("This zombie has a bad knee.")
        }
    }
    required init(town: Town?, monsterName: String) {
        walksWithLimp = false
        isFallingApart = false
        super.init(town: town, monsterName: monsterName)
    }
    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
        super.terrorizeTown()
    }
    ...
```

要实现父类的必需初始化方法，需要在子类的初始化方法实现之前加上`required`关键字。跟覆盖继承自父类的其他方法不同，必需初始化方法不需要用`override`关键字标记，`required`标记已经隐含了覆盖的意思。

这个必需初始化方法的实现使它成为了`Zombie`的指定初始化方法。你也许会问为什么。这是个好问题。

回忆一下，指定初始化方法要负责初始化属性，并且调用父类的初始化方法。这个实现正好做了这两件事情，因此可以调用这个初始化方法来创建`Zombie`实例。

到了这里，你可能很好奇："`Zombie`到底有几个指定初始化方法？"答案是两个：`init(limp:fallingApart:town:monsterName:)`和`init(town:monsterName:)`。有多个指定初始化方法完全没问题，也并不少见。

17.3.4　反初始化

反初始化（deinitialization）是在类的实例没用之后将其清除出内存的过程。从概念上讲，反初始化就是初始化的反面。只有引用类型可以反初始化，值类型不行。

在Swift中，实例被清除出内存之前会调用反初始化方法。这提供了销毁实例前最后做一些维护工作的机会。

内存管理的细节会在第24章详细讨论，不过在讨论初始化的时候介绍一下反初始化的概念也是合理的。

一个类只能有一个反初始化方法。反初始化方法用`deinit`表示，没有参数。代码清单17-17展示了一个`Zombie`的反初始化方法的实际例子。

代码清单17-17　少了一个僵尸（Zombie.swift）

```
    ...
    required init(town: Town?, monsterName: String) {
        walksWithLimp = false
```

```
        isFallingApart = false
        super.init(town: town, monsterName: monsterName)
    }

    deinit {
        print("Zombie named \(name) is no longer with us.")
    }

    final override func terrorizeTown() {
        if !isFallingApart {
            town?.changePopulation(-10)
        }
        super.terrorizeTown()
    }
    ...
```

这个新的反初始化方法只是打印了一句话，跟即将销毁的`Zombie`实例道别。注意，反初始化方法访问了`Zombie`的名字。反初始化方法可以访问实例的所有属性和方法。

打开main.swift，把文件底部的`fredTheZombie`设置为`nil`可以触发`Zombie`的`deinit`方法，将实例从内存中清除。

在Swift中，只有可空类型可以是`nil`。因此，在把`fredTheZombie`设置为`nil`之前要先将其声明为可空——`Zombie?`。这个改动也意味着需要用可空链式调用展开可空实例的值。最后，还需要用`var`而不是`let`来声明`fredTheZombie`，以便实例可以变为`nil`。

代码清单17-18 Fred，我们还没有好好了解你呢（main.swift）

```
...
let var fredTheZombie: Zombie? = Zombie(
    limp: false, fallingApart: false, town: myTown, monsterName: "Fred")
fredTheZombiefredTheZombie?.terrorizeTown()
fredTheZombiefredTheZombie?.town?.printDescription()

var convenientZombie = Zombie(limp: false, fallingApart: false)

print("Victim pool: \(fredTheZombiefredTheZombie?.victimPool)")
fredTheZombiefredTheZombie?.victimPool = 500
print("Victim pool: \(fredTheZombiefredTheZombie?.victimPool)")
print(Zombie.spookyNoise)
if Zombie.isTerrifying {
    print("Run away!")
}
fredTheZombie = nil
```

现在构建并运行程序。你会看到当实例被销毁的时候我们会跟`fredTheZombie`道别。

17.4 可失败的初始化方法

有时候，定义一个初始化可能失败的类型是有用的。在这种情况下，需要一种方法告诉调用者无法初始化实例。我们用*可失败的初始化方法*（failable initalizer）处理这种情况。

有多种原因需要初始化过程失败。第一，初始化方法可能收到了无效参数。举个例子，如果有人试图用负的人口数初始化Town的实例，那就应该让初始化方法失败。第二，初始化可能依赖某个外部资源，而那个资源不可用。例如用代码let image = UIImage(named: "nonexisting-image")创建UIImage会失败，因为图片资源不存在。发生这种情况后，UIImage的可失败的初始化方法会返回nil，表示初始化过程失败了。

Town 的可失败的初始化方法

可失败的初始化方法会返回可空实例。在关键字init后面添加一个问号表示这个初始化方法可能失败（也就是init?）。还可以在init后面添加一个感叹号来创建一个返回隐式展开可空类型的初始化方法（也就是init!）。返回隐式展开可空类型意味着不需要写可空类型展开的语法，Swift利用这些语法使得可空类型更安全。因此，虽然返回隐式可空类型会使初始化方法用起来比较方便，但是安全性会大大下降，所以要谨慎使用。

如果用人口数0初始化Town实例，初始化应该失败，一个镇不能没有人。打开main.swift修改myTown的初始化方法，给population参数传入0。

代码清单17-19　myTown的人口是0（main.swift）

```
var myTown = Town(population: 10_0000, stoplights: 6)
myTown.printDescription()
...
```

这样还不会产生错误。切换到Town.swift为Town结构体添加一个可失败的初始化方法。

Town有两个初始化方法。回忆一下，之前我们把一部分初始化工作从init(population:stoplights:)初始化方法委托给了init(region:population:stoplights:)。现在，只要让init(region:population:stoplights:)可失败就可以了。

代码清单17-20　使用可失败的初始化方法（Town.swift）

```
struct Town {
...
    initinit?(region: String, population: Int, stoplights: Int) {
        guard population > 0 else {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStoplights = stoplights
    }
...
}
```

现在init?(region:population:stoplights:)用了可失败的初始化方法的语法。声明后，检查给定的population值是否小于等于0。如果是，返回nil并且初始化方法失败。在提到可失败的初始化方法时，“失败”的意思是初始化方法会创建一个Town类型的可空实例，其值是nil。这样挺好，值为nil的实例总比属性中有坏数据好。

到这里，应该会出现几个错误。在构建和运行程序前花点时间搞清楚这是怎么回事。

初始化方法`init(population:stoplights:)`目前把一部分工作委托给了一个可失败的初始化方法。这就表示`init(population:stoplights:)`可能会从被委托的初始化方法那里得到`nil`。从被委托的初始化方法那里收到`nil`是出乎意料的，因为`init(population:stoplights:)`本身不会失败。

把`init(population:stoplights:)`改为可失败的可以修复这个问题，如代码清单17-21所示。

代码清单17-21　把Town的两个初始化方法都改为可失败（Town.swift）

```
struct Town {
...
    init?(region: String, population: Int, stoplights: Int) {
        guard population > 0 else {
            return nil
        }
        self.region = region
        self.population = population
        numberOfStoplights = stoplights
    }

    initinit?(population: Int, stoplights: Int) {
        self.init(region: "N/A", population: population, stoplights: stoplights)
    }
...
}
```

运行程序，你会看到还有不少错误要修复。这些错误都位于main.swift中。

`myTown.printDescription()`这行代码有这么一个错误：`Value of optional type 'Town?' not unwrapped; did you mean to use '!' or '?'?`。记住，把`Town`的初始化方法改为可失败的意味着它们会返回可空类型，也就是`Town?`而不是`Town`。这意味着在使用返回值之前需要先展开可空实例。

用可空链式调用修复main.swift中的错误，如代码清单17-22所示。

代码清单17-22　使用可空链式调用（main.swift）

```
var myTown = Town(population: 0, stoplights: 6)
myTownmyTown?.printDescription()
let myTownSize = myTownmyTown?.townSize
print(myTownSize)
myTownmyTown?.changePopulation(1_000_000)
print("Size: \(myTownmyTown?.townSize);
                population: \(myTownmyTown?.population)")
...
```

如你所见，在Swift中表达`nil`会对代码造成相当广泛的影响。这些改动会增加工程复杂度和更多代码，而复杂度和新增代码都会增加犯错的概率。

我们建议只在必要情况下使用可空类型。

现在构建并运行程序。既然修复了错误，工程运行也就没问题了。是时候跟MonsterTown说再见了。下一章我们会采用新的工程，不会再有僵尸了。

17.5　掌握初始化

“我怎么能记得住这么多东西？”我们听到了你的心声。Swift的初始化是一个有大量规则的固定过程。幸运的是，编译器会提醒你需要做什么来遵从规则，写出有效的初始化方法。比起记住初始化的所有规则，从值类型和类这两方面理解Swift的初始化更有用。

对于结构体这样的值类型来说，初始化主要负责确保所有的存储属性都已经被初始化并被赋予了合适的值。这句话对类也适用，只不过类的初始化更复杂一些。

类的初始化过程可以理解为由两个阶段构成。

第一个阶段，类的指定初始化方法最终被调用（无论是直接调用还是通过便捷初始化方法的委托）。到了这里，类声明的所有属性都已经在指定初始化方法中用合适的值初始化了。接着，指定初始化方法把工作委托给父类的指定初始化方法。父类的指定初始化方法又要确保父类的所有存储属性都已经用合适的值初始化了。这是一个持续不断的过程，直到到达继承链的顶端。现在第一个阶段就完成了。

第二个阶段随后开始，为类进一步定制存储属性的值提供了机会。举个例子，指定初始化方法可以在调用父类的指定初始化方法后修改`self`的属性。指定初始化方法还可以调用`self`的实例方法。最后，初始化过程才会进入便捷初始化方法，为其定制实例提供机会。

在这两个阶段之后，实例被完全初始化，所有的属性和方法都可用了。

这个固定的初始化过程的目的在于保证类成功初始化。编译器会确保整个过程的安全；如果你不遵守任意一个步骤，编译器就会报错。因此，只要按照编译器的引导操作，就没有必要记住每一步。随着时间的推移，你对初始化过程的细节会掌握得更好。

17.6　白银挑战练习

现在，`Monster`的必需初始化方法是以`Zombie`子类的指定初始化方法的形式实现的。把这个初始化方法改成`Zombie`的便捷初始化方法。这个改动会涉及把初始化工作委托给`Zombie`的指定初始化方法。

17.7　黄金挑战练习

把任何字符串传递给`monsterName`都可以初始化`Monster`，即使是空字符串也可以，不过会造成`Monster`实例没有名字。虽然弗兰肯斯坦的怪物没有名字，但是你应该会希望能辨认自己的怪物。确保`monsterName`不能为空，以修复`Monster`的这个问题。

解决方法会涉及给`Monster`添加一个可失败的初始化方法。还要注意，这个改动会影响子类`Zombie`的初始化，所以也要对`Zombie`进行必要的修改。

17.8　深入学习：初始化方法参数

跟函数和方法一样，初始化方法可以提供显式的外部参数名。外部参数名用来区分调用者能看到的参数名，而本地参数名则用在初始化方法的实现中。因为初始化方法遵循的命名惯例与函数不同（初始化方法的名字永远是init），所以参数名和类型能用来判断该调用哪个初始化方法。因此，Swift默认为所有的初始化方法参数提供外部名。

你也可以根据需要提供自己的外部参数名。举个例子，假设有一个WeightRecordInLBS结构体，应该可以用千克数来初始化。

```
struct WeightRecordInLBS {
    let weight: Double

    init(kilograms kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

这个初始化方法提供了显式外部参数名kilograms以及本地参数名kilos。在方法实现中，只要把kilos乘以正确的系数就可以将其转换为磅数。这个初始化方法可以这么用：let wr = WeightRecordInLBS(kilograms: 84)。

如果不想暴露外部参数名，甚至可以用_作为显式的外部参数名。举个例子，我们假设的结构体WeightRecordInLBS显然是用磅来定义重量记录的。因此，初始化方法默认用磅数作为参数是合理的。

```
struct WeightRecordInLBS {
    let weight: Double

    init(_ pounds: Double) {
        weight = pounds
    }

    init(kilograms kilos: Double) {
        weight = kilos * 2.20462
    }
}
```

上面新的初始化方法可以这样使用：let wr = WeightRecordInLBS(185)。因为这个类型非常明确地用磅表示重量记录，所以没有必要在参数列表中放一个命名参数。用_可以让代码更紧凑。在调用者明确知道自己传递给参数的是什么值的情况下，这样做比较方便。

第 18 章

值类型与引用类型

18

本章以学习过的关于值类型（比如结构体）和引用类型（比如类）的内容为基础，通过对比一系列场景中两者的不同行为来探索它们的区别。本章结束后，你就应该对何时使用值类型、何时使用引用类型有一个比较好的理解了。

18.1 值语义

创建一个playground，将其命名为ValueVsRefs。playground中应该有下面的模版代码：

```
import Cocoa

var str = "Hello, playground"
```

我们之前见过这段代码很多次了：一个类型为String的可变实例，其值为"Hello, playground"。把str的值赋给另一个实例可以新建一个字符串，如代码清单18-1所示。

代码清单18-1 新建字符串

```
import Cocoa

var str = "Hello, playground"
var playgroundGreeting = str
```

playgroundGreeting的值和str一样，都是字符串"Hello, playground"，通过运行结果侧边栏可以确认这一点。但是，如果改变playgroundGreeting的值会发生什么事？str的值会不会也发生变化？改变playgroundGreeting来寻找答案，如代码清单18-2所示。

代码清单18-2 更新playgroundGreeting

```
import Cocoa

var str = "Hello, playground"
var playgroundGreeting = str
playgroundGreeting += "! How are you today?"
str
```

如你所见，即使playgroundGreeting的值更新了，str的值也不会变。为什么？答案跟值语义（value semantics）有关。

为了更好地理解值语义，按住Option键并点击`playgroundGreeting`，你会看到如图18-1所示的窗口弹出。

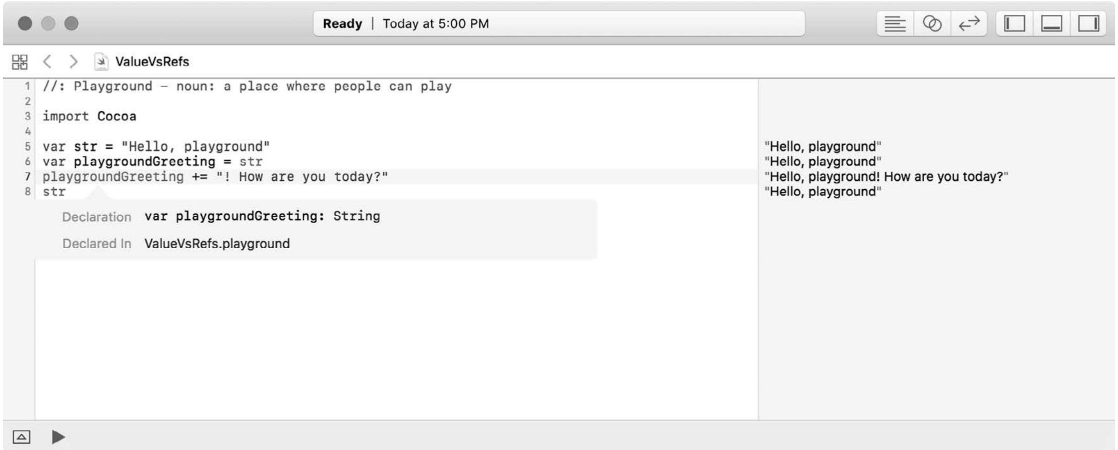


图18-1　`playgroundGreeting`信息

这个弹出窗口显示了一些有用的信息。比如`playgroundGreeting`是`String`类型。点击弹出窗口中的String，它会显示`String`类型的文档（如图18-2所示）。

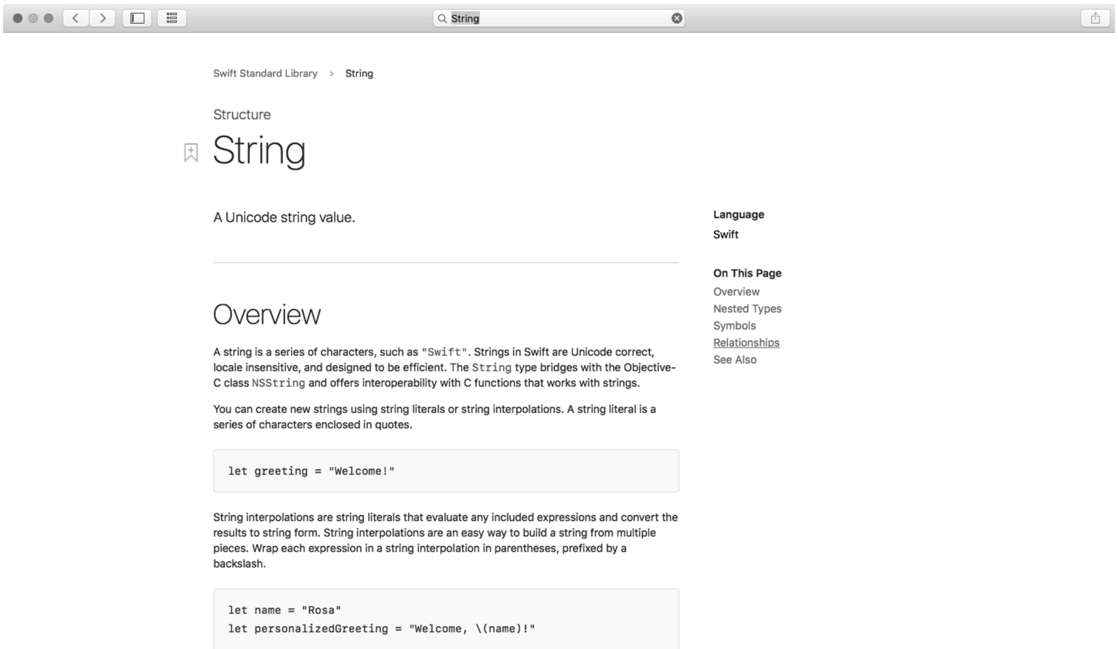


图18-2　`String`文档

在文档的开头你会看到String是struct，这意味着String在Swift标准库中是以struct的形式实现的。更进一步，这意味着String是值类型。在被赋给另一个实例或是作为参数传递给函数时，值类型总是被复制。

在把str赋给playgroundGreeting时，其实是把str的副本赋给了playgroundGreeting。它们没有指向同一个底层实例。因此，修改playgroundGreeting的值不会对str的值造成影响。两者互不相同。图18-3用图示描述了这种关系。

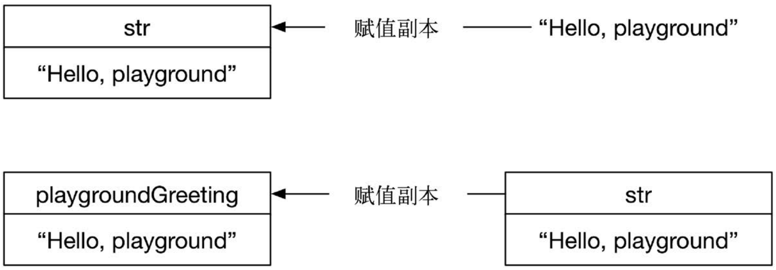


图18-3　值语义和复制行为

Swift的基本类型（Array、Dictionary、Int、String等）都是用结构体实现的，所以都是值类型。在标准库层面选择这种设计应该能让你明白值类型对Swift有多重要。你应该尽量优先用struct实现数据建模，只有在需要的时候才用class。

现在来看看引用语义（reference semantics）的工作原理，以更好地理解什么时候使用这种数据类型。

18.2　引用语义

引用语义跟值语义的行为不同。对于值语义来说，把实例赋给新常量或变量会产生一个副本，把值类型的实例作为参数传递给函数也一样；而引用类型实例的行为则不同，这两种操作会对底层实例创建新的引用（reference）。

给playground增加一个表示希腊神的类，来看看引用语义的意思，如代码清单18-3所示。

代码清单18-3　添加希腊神类

```
import Cocoa

var str = "Hello, playground"
var playgroundGreeting = str
playgroundGreeting += "! How are you today?"
str

class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
```

```
    }
}
```

GreekGod类很小，只提供了一个存储属性用来保存神的名字。创建此类的一个实例，如代码清单18-4所示。

代码清单18-4 创建希腊神

```
...
class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let hecate = GreekGod(name: "Hecate")
```

现在有了一个GreekGod的实例，名为掌管三岔路口的女神Hecate。新建一个常量anotherHecate，把hecate赋给它，如代码清单18-5所示。

代码清单18-5 引用希腊神

```
...
class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let hecate = GreekGod(name: "Hecate")
let anotherHecate = hecate
```

到这里有了两个常量，但是它们都指向GreekGod的同一个实例。改变anotherHecate的名字以说明这一点，如代码清单18-6所示。

代码清单18-6 改变希腊神的名字

```
...
class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let hecate = GreekGod(name: "Hecate")
let anotherHecate = hecate

anotherHecate.name = "AnotherHecate"
anotherHecate.name
hecate.name
```

代码清单18-6中的代码只改变了anotherHecate的名字而没有改变hecate的名字，但是运行

结果侧边栏中两个女神的name属性都变成了"AnotherHecate"。发生了什么？

代码GreekGod(name: "Hecate")创建了一个GreekGod的实例。当把一个类的实例赋给常量或变量时（就像对hecate所做的那样），这个常量或变量会得到实例的引用。如你所见，引用的行为和副本的行为不同。

对于引用来说，常量或变量都指向内存中的同一个实例。因此，hecate和anotherHecate都指向同一个GreekGod的实例。图18-4显示了这种关系。

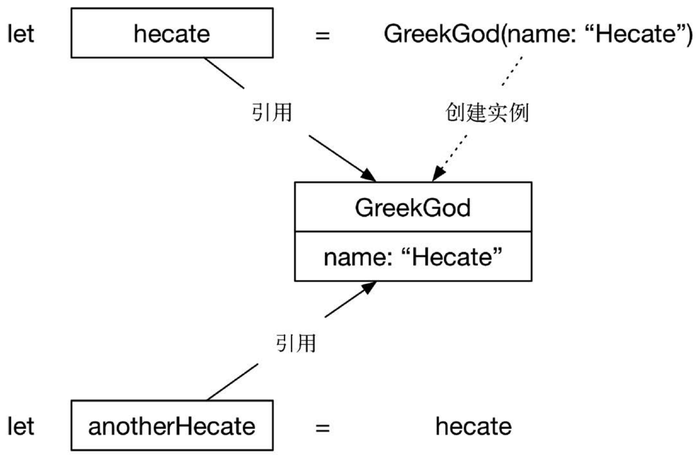


图18-4　引用语义

因为hecate和anotherHecate指向GreekGod的同一个实例，其中一个的改变会反映到另一个上面。

18.3　值类型常量和引用类型常量

值类型常量和引用类型常量的行为不一样。如代码清单18-7所示，创建一个新的结构体Pantheon，这样就有一个我们自己的值类型了。

代码清单18-7　创建希腊神庙

```
...
class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let hecate = GreekGod(name: "Hecate")
let anotherHecate = hecate

anotherHecate.name = "AnotherHecate"
```

```
anotherHecate.name
hecate.name

struct Pantheon {
    var chiefGod: GreekGod
}
```

新结构体表示希腊神庙。它有一个存储属性，表示神庙中居于首位的神。希腊的神永远在争斗，所以这个属性要用var声明。

新建一个Pantheon的实例，把hecate赋给chiefGod，如代码清单18-8所示。

代码清单18-8　Hecate的神庙

```
...
class GreekGod {
    var name: String
    init(name: String) {
        self.name = name
    }
}
let hecate = GreekGod(name: "Hecate")
let anotherHecate = hecate

anotherHecate.name = "AnotherHecate"
anotherHecate.name
hecate.name

struct Pantheon {
    var chiefGod: GreekGod
}

let pantheon = Pantheon(chiefGod: hecate)
```

现在有了一个主神是hecate的Pantheon实例。注意，这个实例是用let创建的，所以是一个常量。试着改变神庙的chiefGod属性，如代码清单18-9所示。

代码清单18-9　新主神

```
...
struct Pantheon {
    var chiefGod: GreekGod
}

let pantheon = Pantheon(chiefGod: hecate)
let zeus = GreekGod(name: "Zeus")
pantheon.chiefGod = zeus
```

首先新建一个GreekGod的实例zeus，接着把这个新实例赋给神庙的chiefGod属性。你会看到那一行有编译错误：Cannot assign to property: 'pantheon' is a 'let' constant。

这个错误告诉我们pantheon是不可变实例，我们无法改变它。声明为常量的值类型不能改变属性，即使属性在类型实现中是用var声明的也是一样。可以把值类型的实例想象成表示一个

整体的值，就像整数一样。如果把整数声明为常量，那么以后就无法改变它的某一部分了。

删除给chiefGod属性赋值的那句代码，以消除编译错误，如代码清单18-10所示。留下zeus，下一个例子会用到它。

代码清单18-10　把Zeus降级

```
...
struct Pantheon {
    var chiefGod: GreekGod
}

let pantheon = Pantheon(chiefGod: hecate)
let zeus = GreekGod(name: "Zeus")
pantheon.chiefGod = zeus
```

引用类型的行为与其不同。试着改变zeus的name属性，如代码清单18-11所示。

代码清单18-11　改变Zeus的名字

```
...
struct Pantheon {
    var chiefGod: GreekGod
}

let pantheon = Pantheon(chiefGod: hecate)
let zeus = GreekGod(name: "Zeus")
zeus.name = "Zeus Jr."
zeus.name
```

这不是*那个*Zeus，而是他众多儿子中的一个。我们用“Jr.”更新了他的名字。虽然zeus是用let声明的，但是编译器对于改名这件事没有意见。

为什么不能改变声明为常量的值类型实例的属性，却能改变声明为常量的引用类型实例的属性呢？

因为zeus是引用类型的实例，所以它指向用代码GreekGod(name: "Zeus")创建的GreekGod的实例。当改变name属性的值时，实际上不是在改变zeus，它只是GreekGod的一个引用。因为在定义GreekGod时name是可变存储属性，所以可以任意改变它。无论改变多少次zeus的名字，zeus指向的都是同一个实例。

18.4　配合使用值类型和引用类型

你可能会产生这样的疑问：“能不能在引用类型内使用值类型？能不能在值类型内使用引用类型？”这两个问题的答案都是“能”。给Pantheon添加类型为GreekGod的属性就已经做到了第二点。尽管我们在引导你这么做的时候没有加以警告，但是在值类型内使用引用类型时要务必小心。（在引用类型内使用值类型倒不会有什么问题。）考虑如代码清单18-12所示例子中对hecate名字的改变。

代码清单18-12　罗马人来了

```
...
let pantheon = Pantheon(chiefGod: hecate)
let zeus = GreekGod(name: "Zeus")
zeus.name = "Zeus Jr."
zeus.name

pantheon.chiefGod.name // "AnotherHecate"
let greekPantheon = pantheon
hecate.name = "Trivia"
greekPantheon.chiefGod.name // ???
```

打印pantheon.chiefGod.name的值时，运行结果侧边栏显示"AnotherHecate"。接着把pantheon的一个副本赋给新常量greekPantheon。记住，因为Pantheon是值类型，所以greekPantheon会得到pantheon的一个副本。然后把hecate的名字改成Trivia。（罗马人打败了希腊人，所以神的名字也都变了。）最后，查看greekPantheon的chiefGod名字，结果出乎意料。

chiefGod的名字现在是Trivia。如果你预期greekPantheon是pantheon的副本，那可能会有点吃惊。这是怎么回事？

记住，chiefGod属性的类型是GreekGod。GreekGod是一个类，所以是引用类型。当我们用hecate作为chiefGod创建pantheon时（Pantheon(chiefGod: hecate)），其实是把一个引用赋给了pantheon的chiefGod。该引用跟hecate指向同一个GreekGod实例。结果就是，改变hecate的name会改变pantheon的chiefGod的name。

这个例子说明了在值类型内使用引用类型的复杂程度。我们预期，当把值类型的实例赋给新变量、常量或传递给函数时，实例会被复制。不过有些令人迷惑的是，一个属性中有引用类型的值类型会把同一个引用传给新变量或新常量。这个引用还是会跟原来的引用指向同一个实例，改变其中任意一个都会反映到所有的引用上。为了避免这种困惑，我们强烈建议大部分情况下都不要在值类型内使用引用类型。如果确实需要在结构体内使用引用类型属性，那么最好使用不可变实例。

18.5　复制

到目前为止，本章的几乎每一个主题中都有创建副本的影子。开发者通常想知道实例复制是浅复制（shallow copy）还是深复制（deep copy）。Swift没有在语言层面提供深复制的支持，这意味着Swift中的复制是浅复制。

为了更好地理解这些概念，来看一个例子。新建一个GreekGod实例，把这个实例和已经存在的实例一起放进一个数组，如代码清单18-13所示。

代码清单18-13　添加一些神

```
...
let athena = GreekGod(name: "Athena")
```

```
let gods = [athena, hecate, zeus]
```

上面的代码新建了希腊神athena，并把这个实例连同hecate和zeus都放进了一个新数组。运行结果侧边栏中会列出包含在数组中的神。

创建数组gods的副本，改变zeus的名字，再比较gods及其副本，如代码清单18-14所示。

代码清单18-14　复制神

```
...
let athena = GreekGod(name: "Athena")

let gods = [athena, hecate, zeus]
let godsCopy = gods
gods.last?.name = "Jupiter"
gods
godsCopy
```

last指向数组的最后一个元素。它是可空类型，因为数组可能为空。运行结果侧边栏看起来类似于图18-5。

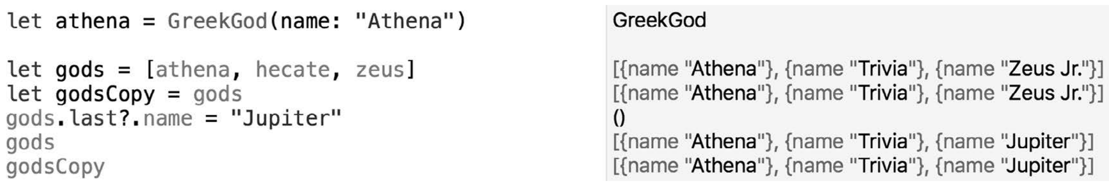


图18-5　比较gods和godsCopy

注意，在改变zeus的name属性后（通过gods.last?.name = "Jupiter"），gods及其副本的内容完全一样。为什么改变gods数组中最后一个神的名字会使得godsCopy中最后一个神的名字也发生变化呢？因为数组是结构体，也就是值类型。就值类型来说，godsCopy是跟gods不同的一个副本。为什么一个数组中一个元素的变化会反映到另一个数组呢？

回忆一下，gods包含的是GreekGod实例。GreekGod是类，也就是引用类型。这意味着godsCopy和gods共享对GreekGod的同一个实例的引用，非常类似于pantheon和greekPantheon共享一个GreekGod实例的引用。

综上所述，last引用gods数组的最后一个元素，就是zeus。当我们改变这个实例的名字时，实际上是在改变zeus指向的GreekGod实例。因此，对zeus的改变会反映到两个数组上。

这种复制被称为浅复制。浅复制不会创建实例的不同副本，而是复制这个实例的引用。图18-6以可视化方式说明了这种行为。

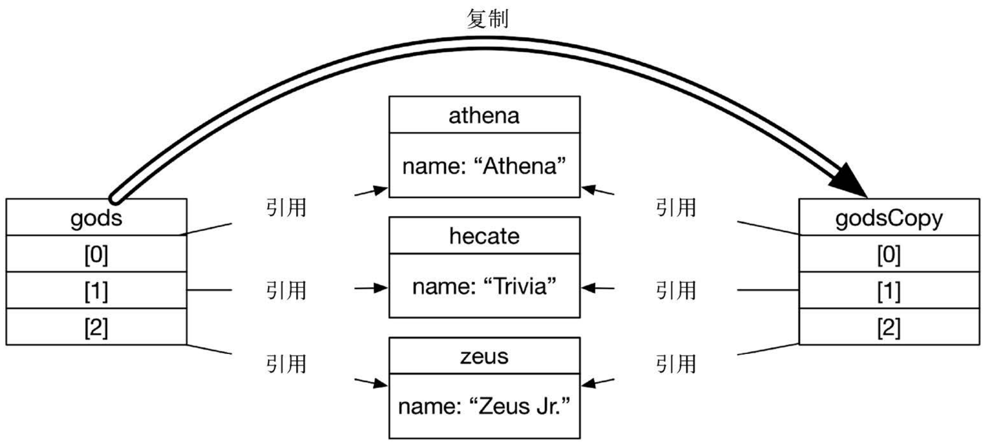


图18-6 `gods`数组的浅复制

深复制会复制引用指向的目标。这意味着`godsCopy`的索引不会引用同一批`GreekGod`的实例。`gods`的深复制会新建一个数组并引用自己的`GreekGod`实例。这种复制看起来如图18-7所示。

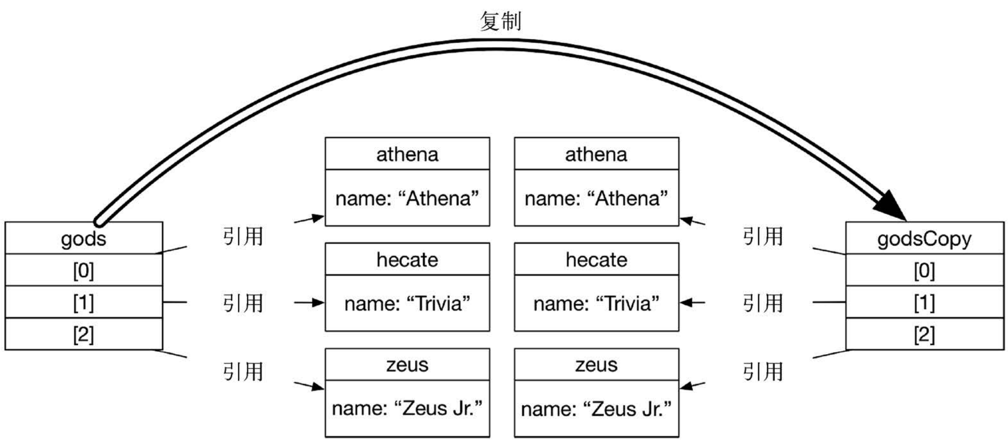


图18-7 `gods`数组的深复制

Swift没有提供执行深复制的方法。如果需要，必须自己编写。

18.6 相等与同一

既然现在理解了值类型和引用类型的区别，就该学习相等和同一了。相等（equality）是指两个实例就可见的特征来说具有一样的值，比如具有同样文本的两个`String`实例。同一（identity）则是指两个变量或常量是否指向内存中的同一个实例。看一下示例代码：

```
let x = 1
let y = 1
x == y // 真
```

我们创建了两个常量，x和y。它们都是Int类型，值都是1。不出所料，使用==的相等性检查得到的结果为真。这是有道理的，因为x和y的值完全一样。

这也是我们想从相等性检查中得到的:两个实例的值是否相等？对于Swift中所有的基本数据类型（String、Int、Float、Double、Array和Dictionary），都可以检查相等性。

athena和hecate都是引用类型，因为它们指向GreekGod实例。因此，可以用同一性运算符（===）检查这两个常量的同一性，从而判断它们是否指向同一个实例。下面是示例代码：

```
athena === hecate // 假
```

同一性检查失败是因为athena和hecate指向内存中的不同位置。

如果检查x和y的同一性会怎么样？你可能以为可以这样使用同一性运算符：x === y。不过这行代码会产生编译错误。为什么？原因是，值类型是传递值的。因为Int在Swift中是用结构体实现的，所以x和y都是值类型。因此，这两个常量不能基于内存中的位置作比较。

如果试图用athena == hecate检查athena和hecate的相等性呢？你会看到编译错误。编译器会告诉你它不知道该如何对GreekGod类调用==函数。如果想对类进行相等性检查，需要通过实现==函数来告诉类该怎么做。这么做会引入协议Equatable，第22章会讲到。

最后有一个重要的注意事项：两个常量或两个变量可能相等（具有相同的值）但不同一（指向给定类型的不同实例）；但是反过来不成立：如果两个变量或常量指向内存中的同一个实例，那它们一定也相等。

18.7　我应该用什么

结构体和类适合用来定义很多自定义类型。Swift出现之前，在OS X和iOS开发中，结构体和类的区别很大，所以两种类型的使用场景都很明确。不过在Swift中，为结构体添加的功能使其行为跟类更接近了。这种相似性使得在什么情况下用哪种类型变得更复杂。

不要绝望。结构体和类之间的重要区别会指引我们选择何时使用哪个。由于要考虑的因素很多，从而很难定义严格的规则，但还是有一些基本指导原则。

(1) 如果类型需要传值，那就用结构体。这么做会确保赋值或传递到函数参数中时类型被复制。

(2) 如果类型不支持子类继承，那就用结构体。结构体不支持继承，所以不能有子类。

(3) 如果类型要表达的行为相对比较直观，而且包含一些简单值，那么考虑优先用结构体实现。有必要的话，之后可以随时把结构体改成类。

(4) 其他所有情况都用类。

结构体在为图形（比如有宽和高的长方形）、区间（比如有起点和终点的赛道）和坐标系中的点（比如二维空间中有*X*和*Y*值的点）建模时比较常用，也很适合定义数据结构：Swift标准库

中的String、Array和Dictionary类型都是用结构体实现的。

还有些其他情况可能用类比结构体合适，不过没有上面的这么常见。举个例子，如果要把引用到处传递，但是又不想有子类，你可能会问自己：到底是用结构体（来避免继承）还是用类（来利用引用语义）？这里的答案是用`final class { ... }`。用final标记类既可以防止其他人继承，又能让实例采用引用语义。

通常来说，除非你绝对清楚自己需要引用类型的哪些好处，否则我们建议一开始用结构体。值类型更容易理解，因为在改变实例副本的值后不需要担心那个实例发生变化。

18.8 深入学习：写时复制

看到这里，你可能会产生一个疑问：Swift对值类型的复制行为会不会对性能产生影响？举个例子，如果每次把一个数组传递给一个函数或者赋给一个常量或变量时都产生一个副本的话，不会产生一大堆没用的副本吗？实际上，这取决于数据及其用法。在实践中，Swift标准库中的值类型实现了被称为写时复制的机制。

写时复制（copy on write，COW）是指对值类型的底层存储的隐式共享。这种优化能够让某个值类型的多个实例共享同一个底层存储，也就是每个实例自己并不持有一份数据的副本；反之，每个实例会维护自己对同一份存储的引用。如果某个实例需要修改或写入存储，那么这个实例就会产生一份自己的副本。这意味着值类型能避免创建多余的数据副本。

为了更好地理解这个概念，我们将实现一个简单的数组来持有Int实例。这个实现能说明这个概念，但并不是Swift数组的实际实现。第22章会更完整地实现一个自定义容器类型。

图18-8展示了IntArray是如何支持COW的。

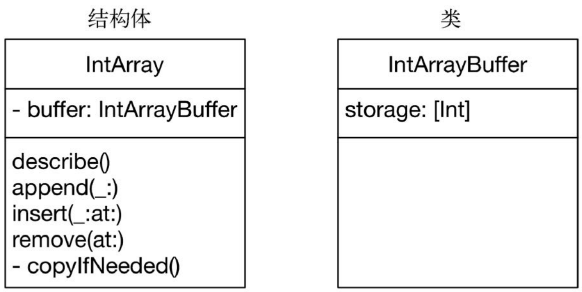


图18-8 IntArray图示

这种图被称为结构图，用到了统一建模语言（unified modeling language，UML）。UML提供了描述和可视化软件工程系统的标准语言。如果图画得好，就能直接照着写代码。举个例子，在IntArray的描述中，buffer和copyIfNeeded()前面的-表示这两个成员是私有的。

IntArray是一个结构体，有一个IntArrayBuffer类型的属性buffer。buffer是私有属性，因为你不想让底层存储对外可见。

IntArrayBuffer是一个类，有一个整数数组类型的属性storage。buffer的类型也是私有的，因为这只是IntArray的实现细节，不需要对外开放。

用整数数组作为存储可能有点奇怪，我们不是在实现能存储整数的数组吗？确实是，不过数组的实际实现会涉及本书后面才会讲到的概念。敬请期待第22章将要讲到的一个真实版本的容器类型。至于现在，最应该关注的是引用类型IntArrayBuffer充当IntArray的底层存储。

IntArray提供属于数组核心功能的三个公开方法：append(_:)、insert(_:at:)和remove(at:)。IntArray还有一个describe()方法，以便观察其底层存储的变化。

写好初始的代码后，我们会给IntArray添加私有的copyIfNeed()方法。这是实现IntArray类型写时复制的地方。

现在可以开始实现数组和底层存储了。创建一个playground，命名为IntArray。首先实现缓存类，如代码清单18-15所示。

代码清单18-15　实现IntArrayBuffer

```
import Cocoa

var str = "Hello, playground"

fileprivate class IntArrayBuffer {
    var storage: [Int]

    init() {
        storage = []
    }

    init(buffer: IntArrayBuffer) {
        storage = buffer.storage
    }
}
```

IntArrayBuffer是一个fileprivate类，这意味着它的实现需要和IntArray位于同一个文件中。因为我们是在playground中写这段代码，所以它对于数组的实现是可见的。

可变存储的类型是[Int]，无参数的初始化方法把它初始化为一个空数组。我们还提供了一个接受IntArrayBuffer实例作为参数的初始化方法。添加这个初始化方法只是为了方便，因为IntArray的实现会用到它。

现在开始写IntArray类型。在IntArrayBuffer类下方创建一个结构体，如代码清单18-16所示。

代码清单18-16　IntArray的第一个实现

```
...
fileprivate class IntArrayBuffer {
    var storage: [Int]

    init() {
        storage = []
```

```
    }

    init(buffer: IntArrayBuffer) {
        storage = buffer.storage
    }
}

struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
    }
}
```

IntArray目前的实现很简单。它有一个私有属性buffer，类型是IntArrayBuffer。这个属性会维护IntArray的后备存储。我们还写了一个没有参数的初始化方法，并利用IntArrayBuffer的空初始化方法设置数组的存储。最后，我们提供了一个describe()方法来打印buffer的storage属性的内容。这样可以追踪数组的变化，以便理解COW的原理。

现在IntArray的实现还很简陋，所做的只是设置buffer作为后备存储。这是很重要的工作，但是数组还缺少主要功能。我们需要写几个方法，实现插入、添加和删除数据，如代码清单18-17所示。

代码清单18-17　定义IntArray的API

```
...
fileprivate class IntArrayBuffer {
    var storage: [Int]

    init() {
        storage = []
    }

    init(buffer: IntArrayBuffer) {
        storage = buffer.storage
    }
}

struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
```

```
    }

    func insert(_ value: Int, at index: Int) {
        buffer.storage.insert(value, at: index)
    }

    func append(_ value: Int) {
        buffer.storage.append(value)
    }

    func remove(at index: Int) {
        buffer.storage.remove(at: index)
    }
}
```

方法insert(_:at:)、append(_:)和remove(at:)都会调用标准库中数组定义的方法。这就强调了用[Int]作为buffer的后备存储的价值。虽然buffer的实现不那么贴合实际，但是至少能简化实现，把关注点聚焦在COW的行为上。

练习使用IntArray，创建一个实例，并添加一些整数（如代码清单18-18所示）。

代码清单18-18　练习使用IntArray

```
...
struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
    }

    func insert(_ value: Int, at index: Int) {
        buffer.storage.insert(value, at: index)
    }

    func append(_ value: Int) {
        buffer.storage.append(value)
    }

    func remove(at index: Int) {
        buffer.storage.remove(at: index)
    }
}

var integers = IntArray()
integers.append(1)
integers.append(2)
integers.append(4)
integers.describe()
```

我们创建了一个IntArray的实例，用append(_:)往数组里添加了几个值。

但是COW的实现还不完整。创建integers的副本，再往里插入一个新的值就能明白为什么了（如代码清单18-19所示）。

代码清单18-19　创建一个IntArray的副本


```
...
struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
    }

    func insert(_ value: Int, at index: Int) {
        buffer.storage.insert(value, at: index)
    }

    func append(_ value: Int) {
        buffer.storage.append(value)
    }

    func remove(at index: Int) {
        buffer.storage.remove(at: index)
    }
}

var integers = IntArray()
integers.append(1)
integers.append(2)
integers.append(4)
integers.describe()
var ints = integers
ints.insert(3, at: 2)
integers.describe()
ints.describe()
```

我们创建了一个IntArray的新实例ints，给它赋值integers。接着，用insert(_:at:)在索引为2的位置插入3来补全这个整数序列。最后，对integers和ints调用describe()来比较它们的storage。

查看控制台就会发现问题。对describe()的两次调用显示integers和ints的storage包含的数据一样。为什么会这样？毕竟我们把IntArray定义为结构体了。结构体应该会被复制，对吧？

问题出在IntArray使用类作为后备存储。根据本章前面的内容，这意味着integers和ints指向同一个引用类型来持有它们的数据。如果其中一个发生了变化，那么另一个也会跟着变化，

因为变化是发生在它们共享的存储上的。

为了解决这个问题，需要确保数组的数据在需要修改时被复制。换句话说，只要不修改共享数据，integers的副本指向同一个底层存储就不会出现问题。为IntArray实现名为copyIfNeeded()的方法来解决问题，如代码清单18-20所示。

代码清单18-20　为IntArray添加COW

```
...
struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
    }

    private mutating func copyIfNeeded() {
        if !isKnownUniquelyReferenced(&buffer) {
            print("Making a copy of \(buffer.storage)")
            buffer = IntArrayBuffer(buffer: buffer)
        }
    }

    func insert(_ value: Int, at index: Int) {
        buffer.storage.insert(value, at: index)
    }

    func append(_ value: Int) {
        buffer.storage.append(value)
    }

    func remove(at index: Int) {
        buffer.storage.remove(at: index)
    }
}

...
print("copying integers to ints")
var ints = integers
print("inserting into ints")
ints.insert(3, at: 2)
...
```

我们添加了一个新方法copyIfNeeded()。把这个方法声明为mutating是因为它会创建一个IntArrayBuffer的新实例并赋给buffer属性。要做到这一点，需要用到我们在上面实现的init(buffer:)初始化方法。这个初始化方法会用和之前缓存一样的数据创建一个新的缓存实例。我们还加了一个print()调用来打印变化信息到控制台。

不过要注意，只有在需要的时候才应该创建新的缓存。条件（if !isKnownUniquely-Referenced(&buffer)）会检查buffer是否只有一个引用。（不用在意buffer是按inout参数的形式传递给这个函数的。这只是实现细节方面的问题；实际上，这个函数不会修改传递给它的参数。）

如果buffer只被引用了一次，那么函数返回true。这种情况下不需要创建新实例，因为没有其他实例在共享底层数据。反之，如果buffer有不止一个引用，那就意味着buffer的storage被多于一个IntArray实例引用了。于是修改buffer就会反映到所有实例上。这种情况下就要创建一个新的buffer。

我们还增加了两个print()调用，把发生的事情打印到控制台。这些打印信息会追踪底层存储什么时候被复制。只有用ints.insert(3, at:2)把新值插入ints中时，buffer才需要被复制。

如果查看控制台，你会发现没有什么变化。integers和ints的buffer还是拥有一样的值。要看到copyIfNeeded()的好处，还需要在mutating方法中调用它。这能讲得通，因为mutating方法会修改实例，所以就需要让这个实例有自己的缓存（如代码清单18-21所示）。

代码清单18-21　为IntArray添加COW，终极版

```
...
struct IntArray {
    private var buffer: IntArrayBuffer

    init() {
        buffer = IntArrayBuffer()
    }

    func describe() {
        print(buffer.storage)
    }

    private mutating func copyIfNeeded() {
        if !isKnownUniquelyReferenced(&buffer) {
            print("Making a copy of \(buffer.storage)")
            buffer = IntArrayBuffer(buffer: buffer)
        }
    }

    mutating func insert(_ value: Int, at index: Int) {
        copyIfNeeded()
        buffer.storage.insert(value, at: index)
    }

    mutating func append(_ value: Int) {
        copyIfNeeded()
        buffer.storage.append(value)
    }

    mutating func remove(at index: Int) {
```

```
            copyIfNeeded()
            buffer.storage.remove(at: index)
        }
    }
    ...
```

注意，我们把insert(_:at:)、append(_:)和remove(at:)标记为了mutating。这些方法现在会修改结构体。这是怎样发生的呢？

要理解的小技巧是，当需要修改一个实例时，就要用新缓存。每次调用一个方法来插入、添加或删除数据时，如果IntArrayBuffer不是被唯一引用的，就需要创建一个新的实例。在这种情况下，copyIfNeeded()会创建一个新的IntArrayBuffer并将其赋给IntArray的buffer属性。于是，通过为这些方法添加copyIfNeeded()调用，我们其实是在告诉编译器，它们可能会修改结构体的属性。（回到第15章查询有关mutating的详细讨论。）

这种策略的优美之处在于，只有当实例需要各不相同时，它们才停止共享一个底层存储。复制一个IntArrayBuffer实例不会导致内存开销，因为副本指向同一个存储，直到发生变化而导致实例需要引用自己的数据为止。如果查看控制台，你会看到integers和ints的内容不再相同。

Swift的容器已经提供了COW支持。你一般不需要自己实现COW类型。举个例子，一个由数字、字典和字符串组成的结构体自动拥有COW，因为它的组成部分都有标准库实现的COW。讨论这部分内容是为了给你COW工作原理的直观感受，并减轻你对值类型的复制行为会带来内存压力的担忧。

Part 5

第五部分

Swift 高级编程

Swift提供了一些高级特性，能让开发者用更复杂的工具来控制应用程序。这部分介绍的概念，对有经验的Swift开发者来说必不可少。协议（protocol）、扩展（extension）和范型（generic）提供的机制，可以帮助人们开发出能充分发挥Swift威力的地道代码。

第 19 章

协　　议

在第16章中，我们学习了利用访问控制来隐藏信息。隐藏信息是封装（encapsulation）的一种形式，可以在设计软件时达到其中一部分的变化不影响其他部分的目的。Swift还提供另一种形式的封装：协议。协议可以让你无须知道类型本身的信息，就能指定并利用类型的接口（interface）。接口是类型提供的一组属性和方法。

协议这个概念比我们目前为止学过的许多主题都抽象。要理解协议及其工作原理，我们将创建一个函数，把数据格式化输出到一个表格中，然后利用协议让这个函数能够灵活处理不同的数据源（data source）。Mac和iOS应用通常把数据的展现和提供数据的源分离。这种分离是一个很有用的模式，让苹果提供处理展现的类，而把决定数据如何存储的工作留给开发者。

19.1　格式化表格数据

新建playground，命名为Protocols。首先声明一个函数，其参数是数组而且该数组的元素本身也是数组（即参数是数组的数组），然后打印字符串到表格中，如代码清单19-1所示。`data`数组的每个元素都是字符串数组，表示一行里的每一列，所以总行数是`data.count`。

代码清单19-1　设置表格

```
import Cocoa

var str = "Hello, playground"

func printTable(_ data: [[String]]) {
    for row in data {
        // 创建空字符串
        var out = "|"

        // 把这一行的每一项都拼接到字符串上
        for item in row {
            out += " \(item) |"
        }

        // 完成，打印出来!
        print(out)
    }
```

```
}

let data = [
    ["Joe", "30", "6"],
    ["Karen", "40", "18"],
    ["Fred", "50", "20"],
]

printTable(data)
```

控制台出现了一个展示数据的简单表格：

```
| Joe | 30 | 6 |
| Karen | 40 | 18 |
| Fred | 50 | 20 |
```


接着，给每一列添加标签，如代码清单19-2所示。列名通过数据单独传入，因为每一列的名字都不一样。

代码清单19-2　给列添加标签

```
...
func printTable(_ data: [[String]], withColumnLabels columnLabels: String...) {
    // 创建包含列头的第一行
    var firstRow = "|"

    for columnLabel in columnLabels {
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
    }
    print(firstRow)

    for row in data {
        // 创建空字符串
        var out = "|"

        // 把这一行的每一项都拼接到字符串上
        for item in row {
            out += " \(item) |"
        }

        // 完成，打印出来!
        print(out)
    }
}

let data = [
    ["Joe", "30", "6"],
    ["Karen", "40", "18"],
    ["Fred", "50", "20"],
]

printTable(data, withColumnLabels: "Employee Name", "Age", "Years of Experience")
```

调试区域的第一行出现的应该是列标签。

```
| Employee Name | Age | Years of Experience |
| Joe | 30 | 6 |
| Karen | 40 | 18 |
| Fred | 50 | 20 |
```

列的宽度各不相同，所以这个表格很难看。记录每一列标签的宽度，然后为数据元素填充空格就可以解决这个问题，如代码清单19-3所示。

代码清单19-3　对齐各列

```
...
func printTable(_ data: [[String]], withColumnLabels columnLabels: String...) {
    // 创建包含列头的第一行
    var firstRow = "|"

    // 记录每一列的宽度
    var columnWidths = [Int]()

    for columnLabel in columnLabels {
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(columnLabel.characters.count)
    }
    print(firstRow)

    for row in data {
        // 创建空字符串
        var out = "|"

        // 把这一行的每一项都拼接到字符串上
        for item in row {
            out += " \(item) |"
        for (j, item) in row.enumerated() {
            let paddingNeeded = columnWidths[j] - item.characters.count
            let padding = repeatElement(" ", count:
                          paddingNeeded).joined(separator: "")
            out += " \(padding)\(item) |"
        }

        // 完成，打印出来！
        print(out)
    }
}
...
```

在构建第一行时，要把每个列头的宽度记录到`columnWidths`数组中。接着，在把每个数据项拼接到输出行时，计算数据项比列头短多少，并存入`paddingNeeded`。然后用`paddingNeeded`个空格构建一个字符串，方法是调用`repeatElement(_:count:)`函数。这个函数会创建一组空格，再调用容器类型的`joined(separator:)`方法把这些空格拼接成一个字符串。

查看调试区，现在有一个格式漂亮的数据表格了。

```
| Employee Name | Age | Years of Experience |
|           Joe |  30 |                   6 |
|         Karen |  40 |                  18 |
|          Fred |  50 |                  20 |
```

不过，printTable(_:withColumnLabels:)函数至少还有一个大问题：很难用！需要有单独的数组提供列标签和数据，还得手动确保列标签的数量和数据数组的元素数量一样。

这种信息最好用结构体和类表示。把调用printTable(_:withColumnLabels:)的部分代码换成模型对象（model object），它们表示应用程序用到的数据的类型，如代码清单19-4所示。

代码清单19-4　使用模型对象

```
...
let data = [
    ["Joe", "30", "6"],
    ["Karen", "40", "18"],
    ["Fred", "50", "20"],
]

printTable(data, withColumnLabels: "Employee Name", "Age", "Years of Experience")
struct Person {
    let name: String
    let age: Int
    let yearsOfExperience: Int
}

struct Department {
    let name: String
    var people = [Person]()

    init(name: String) {
        self.name = name
    }

    mutating func add(_ person: Person) {
        people.append(person)
    }
}

var department = Department(name: "Engineering")
department.add(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.add(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.add(Person(name: "Fred", age: 50, yearsOfExperience: 20))
```

现在有了Department，你想用printTable(_:withColumnLabels:)打印出每个人的详情。可以修改函数，使其接受一个Department而不是现在的两个参数。不过，现在的printTable(_:withColumnLabels:)可以用来打印任何表格式的数据，如果能保留这个特性就好了。下面介绍的协议可以帮助保留这个功能。

19.2　协议

协议(protocol)能让你定义类型需要满足的接口。满足某个协议的类型被称为符合(conform)这个协议。

定义一个协议，指定`printTable(_:withColumnLabels:)`所需的接口，如代码清单19-5所示。这个函数需要知道有几行几列，每列的标签是什么，每个单元要展示的数据项是什么。对于Swift编译器来说，把这个协议放在playground中的哪个地方都无所谓。不过放在`printTable(_:withColumnLabels:)`前面可能最合理，因为这个函数要用到该协议。

代码清单19-5　定义协议

```
...
protocol TabularDataSource {
    var numberOfRows: Int { get }
    var numberOfColumns: Int { get }

    func label(forColumn column: Int) -> String

    func itemFor(row: Int, column: Int) -> String
}

func printTable(_ data: [[String]], withColumnLabels columnLabels: String...) {
    ...
}
...
```

协议的语法看起来很熟悉，跟定义结构体或类的语法很像，不过所有的计算属性和函数定义都被省略了。`TabularDataSource`协议指出，符合这个协议的类型都必须有两个属性：`numberOfRows`和`numberOfColumns`。`{ get }`语法表示这些属性可读。如果属性需要被读写，那就要用`{ get set }`。注意，把协议的属性标记为`{ get }`并不能排除符合协议的类型有可读写属性的可能，只是表示这个协议需要让这个属性可读。最后，`TabularDataSource`指定符合协议的类型必须拥有`label(forColumn:)`和`itemFor(row:column:)`这两个方法，类型如上面所列。

协议定义类型必须拥有的一组最少的属性和方法。除去协议所列出的，类型也可以有更多的属性和方法，只要协议的需求被满足即可。

让`Department`符合`TabularDataSource`协议。先从声明它符合这个协议开始，如代码清单19-6所示。

代码清单19-6　声明Department符合TabularDataSource

```
...
struct Department: TabularDataSource {
...
```

符合协议的语法是在类型后面添加`: ProtocolName`。（看起来有点像声明父类。我们会讲到

如何一起使用协议和父类。）

现在playground有个错误。点击错误图标查看详情。我们声明了`Department`符合`TabularDataSource`，但是`Department`没有`TabularDataSource`所需的属性和方法。把它们的实现都加上，如代码清单19-7所示。

代码清单19-7 添加所需的属性和方法

```
...
struct Department: TabularDataSource {
    let name: String
    var people = [Person]()

    init(name: String) {
        self.name = name
    }

    mutating func add(_ person: Person) {
        people.append(person)
    }

    var numberOfRows: Int {
        return people.count
    }

    var numberOfColumns: Int {
        return 3
    }

    func label(forColumn column: Int) -> String {
        switch column {
        case 0: return "Employee Name"
        case 1: return "Age"
        case 2: return "Years of Experience"
        default: fatalError("Invalid column!")
        }
    }

    func itemFor(row: Int, column: Int) -> String {
        let person = people[row]
        switch column {
        case 0: return person.name
        case 1: return String(person.age)
        case 2: return String(person.yearsOfExperience)
        default: fatalError("Invalid column!")
        }
    }
}
...
```

`Department`针对每个人都有一行，所以`numberOfRows`属性会返回部门的人数。每个人有三个需要展示的属性，所以`numerOfColumns`返回3。每行的标签是人名。`label(forColumn:)`和

itemFor(row:column:)比较有意思：用switch语句返回两个列头中的一个。（为什么有default分支？如果不确定的话就回去看看第5章。）

现在Department符合了TabularDataSource协议，playground中的错误就消失了。不过，还是要改一下printTable(_:withColumnLabels:)，让它能接受并配合TabularDataSource，因为现在没有办法调用这个函数并传递department。协议不仅能定义符合该协议的类型必须提供的属性和方法，自己还能作为类型使用：变量、函数参数和返回值都可以把协议作为类型。

既然这个协议提供了和旧参数一样的数据（包括所有的列标签和数据量），那就把printTable(_:withColumnLabels:)改成接受类型为TabularDataSource的数据源，如代码清单19-8所示。

代码清单19-8　让printTable(_:)接受TabularDataSource

```
...
func printTable(_ data: [[String]], withColumnLabels columnLabels: String...) {
func printTable(_ dataSource: TabularDataSource) {
    // 创建包含列头的第一行
    var firstRow = "|"

    // 记录每一列的宽度
    var columnWidths = [Int]()

    for columnLabel in columnLabels {
    for i in 0 ..< dataSource.numberOfColumns {
        let columnLabel = dataSource.label(forColumn: i)
        let columnHeader = " \(columnLabel) |"
        firstRow += columnHeader
        columnWidths.append(columnLabel.characters.count)
    }
    print(firstRow)

    for row in data {
    for i in 0 ..< dataSource.numberOfRows {
        // 创建空字符串
        var out = "|"

        // 把这一行的每一项都拼接到字符串上

        for (j, item) in row.enumerated() {
            for j in 0 ..< dataSource.numberOfColumns {
            let item = dataSource.itemFor (row:i, column:j)
            let paddingNeeded = columnWidths[j] - item.characters.count
            let padding = repeatElement(" ", count:
                           paddingNeeded).joined(separator: "")
            out += " \(padding)\(item) |"
        }

        // 完成，打印出来!
        print(out)
    }
}
...
```

现在Department符合了TabularDataSource，而且printTable(_:)被改成了能够接受TabularDataSource。因此，可以打印部门信息了。添加对printTable(_:)的调用，如代码清单19-9所示。

代码清单19-9 打印Department

```
...
var department = Department(name: "Engineering")
department.addPerson(Person(name: "Joe", age: 30, yearsOfExperience: 6))
department.addPerson(Person(name: "Karen", age: 40, yearsOfExperience: 18))
department.addPerson(Person(name: "Fred", age: 50, yearsOfExperience: 20))

printTable(department)
```

确认调试区的输出仍然能反映department的结构：

```
| Employee Name | Age | Years of Experience |
|           Joe |  30 |                   6 |
|         Karen |  40 |                  18 |
|          Fred |  50 |                  20 |
```


19.3 符合协议

如前所述，符合协议的语法看起来跟第15章的声明父类完全一样。这就带来了如下几个问题。

(1) 哪些类型可以符合协议？

(2) 一个类型可以符合多个协议吗？

(3) 一个类可以在有父类的同时符合协议吗？

所有的类型都可以符合协议。刚才我们让一个结构体（Department）符合了一个协议。枚举和类也可以符合协议。声明枚举符合协议的语法跟结构体完全一样：类型声明后跟一个冒号和协议名。（类稍微复杂一些，我们稍后会讲到。）

一个类型也可以符合多个协议。CustomStringConvertible是Swift定义的一个协议，当类型希望控制自己的实例被如何转换为字符串时就可以实现这个协议。诸如print()的其他函数会在判断如何显示要打印的值时检查它们是否符合CustomStringConvertible。CustomString-Convertible只有一个要求：类型必须有一个返回String的可读属性description。修改Department让它符合TabularDataSource和CustomStringConvertible，用逗号隔开协议，如代码清单19-10所示。

代码清单19-10 符合CustomStringConvertible

```
...
struct Department: TabularDataSource, CustomStringConvertible {
    let name: String
    var people = [Person]()

    var description: String {
        return "Department (\(name))"
```

```
    }
...
}
```

这里把description实现为只读计算属性。现在打印部门的时候就可以看到它的名字了，如代码清单19-11所示。

代码清单19-11　打印部门的名字

```
...
printTable(department)
print(department)
```

最后，类也可以符合协议。如果类没有父类，语法就跟结构体和枚举一样：

```
class ClassName: ProtocolOne, ProtocolTwo {
    // ...
}
```

如果类有父类，那么父类的名字在前，后跟协议（或者多个协议）。

```
class ClassName: SuperClass, ProtocolOne, ProtocolTwo {
    // ...
}
```

19.4　协议继承

Swift支持协议继承（protocol inheritance）。继承另一个协议的协议要求符合的类型提供它本身及其所继承协议的所有属性和方法。这跟类的继承不同：类的继承定义的是父类和子类之间的紧密联系，而协议继承只是把父协议的需求添加到子协议上。举个例子，修改TabularDataSource，让它继承CustomStringConvertible协议，如代码清单19-12所示。

代码清单19-12　让TabularDataSource继承CustomStringConvertible

```
protocol TabularDataSource: CustomStringConvertible {
    var numberOfRows: Int { get }
    var numberOfColumns: Int { get }

    func label(forColumn column: Int) -> String

    func itemFor(row: Int, column: Int) -> String
}
...
```

现在，所有符合TabularDataSource的类型也必须符合CustomStringConvertible。这意味着类型必须提供TabularDataSource的所有属性和方法，外加CustomStringConvertible所需的description属性。利用这一点在printTable(_:)中打印表头。现在不需要代码清单19-11添加的print()调用了，所以把它删除，如代码清单19-13所示。

代码清单19-13　打印表头

```
...
func printTable(dataSource: TabularDataSource) {
    print("Table: \(dataSource.description)")
    ...
}
...
printTable(department)
print(department)
```

现在，调试区的打印信息包含了对表格的描述：

```
Table: Department (Engineering)
| Employee Name | Age | Years of Experience |
|           Joe |  30 |                   6 |
|         Karen |  40 |                  18 |
|          Fred |  50 |                  20 |
```

协议还可以继承多个其他协议，就像类型可以符合多个协议一样。继承多个协议的语法你大概能猜到——用逗号分隔多出来的协议，就像这样：

```
protocol MyProtocol: MyOtherProtocol, CustomStringConvertible {
    // MyProtocol的需求
}
```

19.5　协议组合

协议继承是一个强大的工具，能让我们基于现有的一个或一组协议通过添加需求来方便地新建协议。不过，使用协议继承可能会让你在创建类型时作出错误的决策。事实上，这正是让TabularDataSource继承CustomStringConvertible会产生的问题，因为我们希望能够打印数据源的描述。CustomStringConvertible与表格数据源没有任何内在联系。回去修复这个打印数据源的错误尝试，如代码清单19-14所示。

代码清单19-14　TabularDataSource不应该是CustomStringConvertible

```
protocol TabularDataSource: CustomStringConvertible {
...
}
```

现在，如果我们尝试获取传递给printTable(_:)的数据源的description，编译器理所当然地开始抱怨了。可以用协议组合（protocol composition）解决这个问题，不需要用转换字符串这样的无关需求来污染TabularDataSource，如代码清单19-15所示。

代码清单19-15　让printTable的参数符合CustomStringConvertible

```
...
func printTable(dataSource: TabularDataSource & CustomStringConvertible) {
    print("Table: \(dataSource.description)")
    ...
}
```

协议组合的语法用关键字&中缀操作符告诉编译器，我们把多个协议组合成了单个的需求。协议组合可以对多于两个的协议使用，只要用逗号分隔、放在尖括号（<>）中即可。上面的例子需要dataSource同时符合TabularDataSource和CustomStringConvertible。

考虑另一种可能性。可以新建一个同时继承TabularDataSource和CustomStringConvertible的协议，就像这样：

```
protocol PrintableTabularDataSource: TabularDataSource, CustomStringConvertible {
}
```

然后我们就能把这个协议用作printTable(_:)的参数类型了。PrintableTabularDataSource和TabularDataSource & CustomStringConvertible都要求符合它们的类型实现TabularDataSource和CustomStringConvertible需要的所有属性和方法。那么两者有什么区别？

区别是PrintableTabularDataSource是一个单独的类型。要使用这个类型，必须修改Department，声明它符合PrintableTabularDataSource——即使它已经符合了所有的需求。另一方面，protocol<TabularDataSource, CustomStringConvertible>不会创建新类型，它只是表示printTable(_:)的参数同时符合这两个协议。因此，不需要回去修改Department。因为它已经符合了TabularDataSource和CustomStringConvertible，所以也符合TabularDataSource & CustomStringConvertible。

19.6　mutating 方法

回忆一下，第14章和第15章介绍了值类型（结构体和枚举）的方法不能修改self，除非这个方法被标记为mutating。协议的方法默认是nonmutating。在第14章的Lightbulb枚举中，toggle()方法为mutating。

```
enum Lightbulb {
    case on
    case off
    ...
    mutating func toggle() {
        switch self {
        case .on:
            self = .off

        case .off:
            self = .on
        }
    }
}
```

假设现在我们想定义一个协议，表示一个实例可以开关：

```
protocol Toggleable {
    func toggle()
}
```

声明Lightbulb符合Toggleable会造成编译错误。错误消息中有注解详述了这个问题：

```
error: type 'Lightbulb' does not conform to protocol 'Toggleable'

note: candidate is marked 'mutating' but protocol does not allow it
mutating func toggle() {
              ^
```

注解指出Lightbulb中的toggle()方法被标记为mutating，但是Toggleable协议期望的是nonmutating的函数。在协议定义中把toggle()标记为mutating可以修复这个问题：

```
protocol Toggleable {
    mutating func toggle()
}
```


一个符合Toggleable协议的类不需要标记toggle()方法为mutating。类的方法总是可以改变self的属性，因为类是引用类型。

19.7　白银挑战练习

printTable(_:)函数有个bug：如果有数据项比所在列的标签长，它就会崩溃。试着把Joe的年龄改成1000来试试看。修复这个bug。（简单的修复是让函数不崩溃。较难的修复是让表格所有的行和列仍然正确对齐。）

19.8　黄金挑战练习

新建类型BookCollection，使其符合TabularDataSource。对书单调用printTable(_:)应该显示书的表格，列是书名、作者和亚马逊的平均评分。（除非所有书的书名和作者名字都很短，否则你必须先完成上一个练习！）

第 20 章

错误处理

20

你用过的软件是否经常崩溃或者做一些出乎你意料的事情？大部分情况下，这些问题是不正确的错误处理造成的。错误处理是软件开发中的幕后英雄：没人认为它重要；如果做好了，没人会注意到。同时它又非常关键：如果没做好，软件的用户肯定会注意到（而且会抱怨）。本章会探索Swift提供的捕获和处理错误的工具。

20.1 错误分类

可能发生的错误分为两大类：可恢复的错误（recoverable error）和不可恢复的错误（nonrecoverable error）。

可恢复的错误通常是我们必须准备好面对并进行处理的事件。可恢复的错误有下面几个常见例子：

- 试图打开不存在的文件；
- 试图和下线的服务器通信；
- 试图在设备没有网络连接时通信。

Swift为处理可恢复的错误提供了丰富的工具。你可能已经习惯Swift在编译期的强制安全规则了，但是错误处理是另一码事。在调用一个可能以可恢复错误的形式失败的函数时，Swift需要你知道并处理这种可能性。

不可恢复的错误只是一类特殊的bug。前面已经遇到过一种了：强制展开值为`nil`的可空实例。还有一个例子是试图越过数组边界访问元素。这些不可恢复的错误程序会触发陷阱（trap）。

第4章提到，当程序触发陷阱时会马上停止执行。陷阱是操作系统的底层命令，用来立即中断程序执行。如果程序是从Xcode运行，会停在调试器中并告诉你哪里出错了。不过，对于运行程序的用户来说，陷阱就跟崩溃一样——程序立即退出。

Swift为什么要对这类错误这么严格呢？顾名思义，这类错误不可恢复，也就是说程序无法做什么来修复问题。举个例子，考虑展开可空实例。当强制展开可空实例时，我们期望得到一个值，而其他的代码也都假设可以利用这个值。如果可空实例是`nil`，那就没有值。Swift能做的唯一合理的事情就是停止程序。如果程序继续运行，就可能会在访问不存在的值时崩溃；更糟糕的情况是，程序可能会继续运行但是产生错误的结果。（这两种情况都会在安全性较低的语言中出现，比如C。）

本章会构建一个简单的两步编译器。在这个过程中，我们会实现一个函数，用来计算基本的数学表达式。举个例子，提供输入字符串"10 + 3 + 5"，函数会返回整数18。随着学习的深入，我们会用到Swift中用来处理可恢复错误和不可恢复错误的设施。

20.2　对输入字符串做词法分析

表达式计算编译器的第一步是**词法分析**（lexing）。词法分析是把输入转化为一个**符号**（token）序列。符号就是某种有意义的东西，比如数或者加号（编译器认识的两种符号）。词法分析有时候也被称为“符号化”，因为我们是把某种对编译器来说无意义的输入（比如字符串）变成有意义的符号序列。

新建playground，名为ErrorHandling。定义一个有两个成员的枚举，分别对应两种符号，如代码清单20-1所示。

20

代码清单20-1　声明Token类型

```
import Cocoa

~~var str = "Hello, playground"~~

enum Token {
    case number(Int)
    case plus
}
```

接着，开始构建分析器。要分析输入字符串，需要一个一个地访问输入字符串的每个字符。还需要记录当前在字符集合中的位置。创建Lexer类，给它两个属性，如代码清单20-2所示。

代码清单20-2　创建Lexer

```
import Cocoa

enum Token {
    case number(Int)
    case plus
}

class Lexer {
    let input: String.CharacterView
    var position: String.CharacterView.Index

    init(input: String) {
        self.input = input.characters
        self.position = self.input.startIndex
    }
}
```

第7章中提到，每个字符串都有一个characters属性，它是Character的集合。characters属性的类型是String.CharacterView。每个String.CharacterView都有startIndex和

endIndex属性，可以用来遍历字符。把input属性初始化为输入字符串的characters属性，把position属性初始化为字符视图的起点。

对输入字符做词法分析是一个简单的过程。要实现的步骤简要展示在图20-1中。

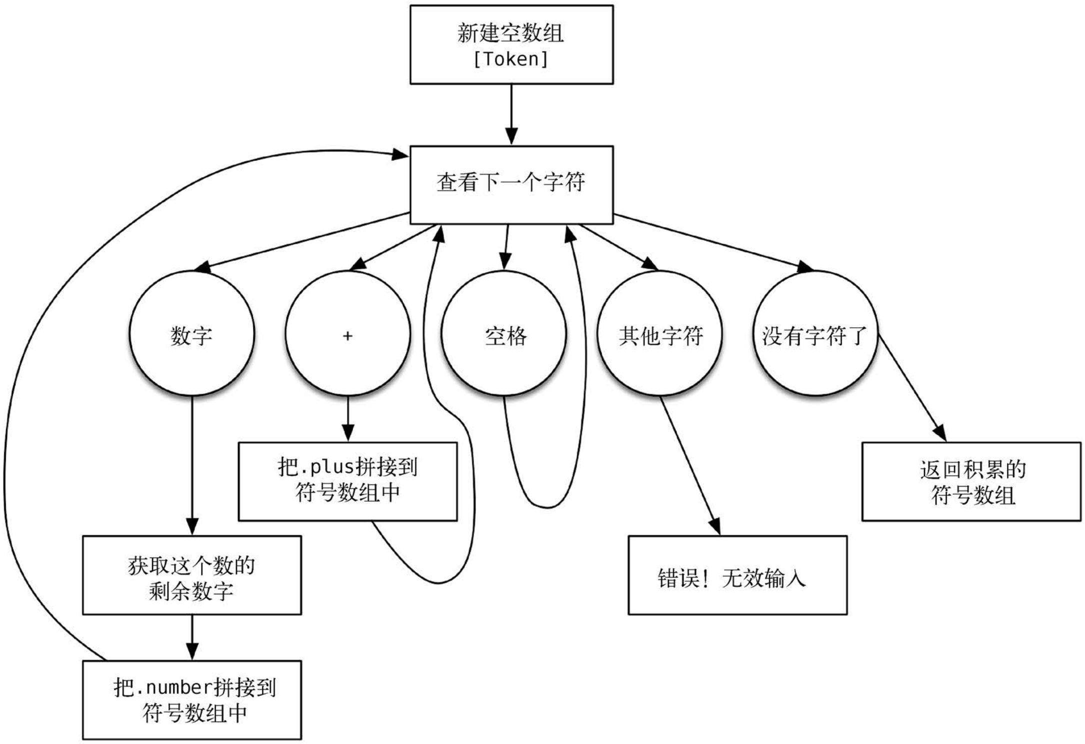


图20-1　词法分析算法

首先创建一个空数组来保存Token实例。接着查看第一个字符。如果是数字（digit，即一个0~9的数），那就继续往前扫描输入字符串，收集后面所有的数字形成一个数，再往符号数组里添加一个.number。如果字符是+，就往Token数组中添加一个.plus。如果字符是空格，忽略它。以上三种情况发生后，都要在输入字符串中前进一个字符，重复上面的决策过程。在扫描输入字符串的过程中，无论什么时候遇到上述三种情况以外的字符，都认为遇到了一个错误：输入无效。当到达输入字符串的末尾时，词法分析阶段就结束了。

要实现这个算法，Lexer需要两个基本操作：从输入中查看下一个字符，以及从当前位置前进一个字符。查看下一个字符的操作需要以某种方式表明分析器已经到达输入的终点，因此让它返回可空类型，如代码清单20-3所示。

代码清单20-3　实现peek()

```
...
```

```
class Lexer {
    let input: String.CharacterView
    var position: String.CharacterView.Index

    init(input: String) {
        self.input = input.characters
        self.position = self.input.startIndex
    }

    func peek() -> Character? {
        guard position < input.endindex else {
            return nil
        }
        return input[position]
    }
}
```


用guard语句确保没有到达输入的终点，如果到达了就返回nil。如果还有输入，就返回当前位置的字符。

现在分析器可以查看当前的字符了，还需要前进到下一个字符的方法。前进很简单，position是输入字符集的索引，每个字符集都知道如何相对于某个旧索引计算新索引。用input的index(after:)方法得到position当前值的下一个索引再赋给position。如代码清单20-4所示。

代码清单20-4　实现advance()

```
...
class Lexer {
    ...
    func peek() -> Character? {
        guard position < input.endindex else {
            return nil
        }
        return input[position]
    }

    func advance() {
        position = input.index(after: position)
    }
}
```

在继续之前，这里有个机会介绍如何检查不可恢复的错误。在实现Lexer的剩余部分时会调用到peek()和advance()。peek()可以在任何时候调用，但是advance()只有在还没到达输入终点的情况下才能调用。给advance()添加一个断言（assertion）来检查这种情况，如代码清单20-5所示。

代码清单20-5　给advance()添加断言

```
...
class Lexer {
```

```
    ...
    func advance() {
        assert(position < input.endIndex, "Cannot advance past endIndex!")
        position = input.index(after: position)
    }
}
```

assert(_:_:)有什么用？它的第一个参数是要检查的条件。如果条件计算为true，什么都不会发生。不过，如果条件计算为false，程序就会触发陷阱，进入调试器，显示第二个参数提供的消息。

对assert(_:_:)的调用只有在程序用调试模式构建时才会生效。在playground中写代码或者从Xcode中运行工程时默认使用调试模式。如果是为了提交到App Store而构建应用，Xcode使用发布模式。在发布模式构建的一个不同点是，这会打开一系列编译器优化并删除所有assert(_:_:)调用。

如果想在发布模式中保留断言，就用precondition(_:_:)代替。它的参数、效果都和assert(_:_:)一样，但是不会在以发布模式构建时被删除。

为什么要用assert(_:_:)而不是guard或者其他错误处理机制呢？assert(_:_:)及其搭档precondition(_:_:)是用来捕获不可恢复的错误的工具。在实现词法分析算法的时候，我们会不断从输入的开头到结束移动索引。我们不应该试图前进到超出input的endIndex。加上这个断言可以找到可能引入这类bug而产生的任何错误，因为断言会导致调试器在这个点停止执行代码，帮助我们定位错误。如果不用断言，要么词法分析器在输入流中的位置不再前进，要么把错误返回给词法分析器的调用者；无论哪种做法都不合理。

现在Lexer有了我们需要的构件，就可以开始实现词法分析算法了。词法分析的输出是Token数组，但是词法分析也可能失败。为了表示一个函数或方法可能抛出错误，在包含参数的圆括号后添加throws关键字，如代码清单20-6所示。（lex()的这个实现并不完整而且无法编译，不过我们很快就会完成它。）

代码清单20-6　声明抛出错误的lex()方法

```
...
class Lexer {
    ...
    func advance() {
        assert(position < input.endIndex, "Cannot advance past endIndex!")
        position = input.index(after: position)
    }

    func lex() throws -> [Token] {
        var tokens = [Token]()

        while let nextCharacter = peek() {
            switch nextCharacter {
            case "0" ... "9":
                // 开始处理数，需要获取数的剩余部分
```

```
                case "+":
                    tokens.append(.plus)
                    advance()

                case " ":
                    // 前进，忽略空格
                    advance()

                default:
                    // 出乎意料——需要返回错误
                }
            }

            return tokens
        }
    }
```


现在已经实现了词法分析算法的大部分。首先从创建一个数组tokens开始，用来保存分析出来的每个Token。用while let条件循环一直到达输入的末尾。对于每个字符，都会进入四个分支的其中之一。我们已经实现了字符为加号（把.plus拼接到tokens，并前进到下一个字符）和空格（忽略并前进到下一个字符）的情况。

还有两种情况要实现。先从default分支开始。如果分支匹配，那么下个字符是预期之外的。这意味着需要**抛出**（throw）一个错误。在Swift中，用throw关键字来发送（抛出）错误给调用者。

可以抛出什么呢？必须抛出符合Error协议的类型的实例。大多数时候，要抛出的错误都定义为枚举，但仍然符合这个协议。

符合Error协议的枚举应该叫什么名字呢？一个选择是叫LexerError。我们可以接受用LexerError做名字，但这样仅仅为了命名词法分析错误就引入了一种新类，并不理想。一个独立的类型会让人觉得LexerError在Lexer以外还有用。回忆第16章的内容，我们可以用嵌入类型。可以用嵌入的Lexer.Error枚举，这样能很容易看出来它提供的错误成员是跟Lexer直接相关的。

在Lexer内部声明一个枚举，用来表示词法分析错误，如代码清单20-7所示。

代码清单20-7 声明Lexer.Error

```
...
class Lexer {
    enum Error {
        case invalidCharacter(Character)
    }

    let input: String.CharacterView
    var position: String.CharacterView.Index
    ...
}
```

Lexer.Error需要符合Error协议。直接尝试添加协议会失败，试试看会出现什么样的错误（如代码清单20-8所示）。

代码清单20-8　尝试让Lexer.Error符合Error

```
...
class Lexer {
    enum Error: Error {
        case invalidCharacter(Character)
    }

    let input: String.CharacterView
    var position: String.CharacterView.Index
    ...
}
```

编译器的报错（如图20-2所示）可能不好理解。

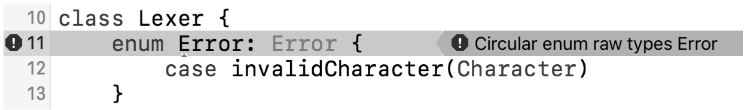


图20-2　尝试让Lexer.Error符合Error会发生错误

这里要注意的关键词是circular enum。编译器之所以犯糊涂，是因为它认为我们在试图用Error类型定义其本身。那如何才能告诉编译器我们想用Swift标准库中的Error协议呢？

跟C++不同，Swift没有显式的命名空间，而是让所有的类型和函数隐式地处于它们所在模块的命名空间。一般不需要关心当前是在哪个模块中写代码。比如，如果是写iOS应用，整个应用位于一个模块中。

声明Lexer.Error是少数几种需要注意模块的情形之一。作为Swift标准库的一部分，类型和函数位于Swift模块中。我们可以用全名Swift.Error来指定使用Swift模块中的Error类型。加上全名来修复声明Lexer.Error的问题，如代码清单20-9所示。

代码清单20-9　让Lexer.Error符合Error

```
...
class Lexer {
    enum Error: Swift.Error {
        case invalidCharacter(Character)
    }
    ...
}
```

按住Command键并点击Swift.Error协议来查看它在Swift标准库中的定义。你会发现这是一个空协议。也就是说，它不需要任何属性或方法。只要如此声称，任何类型都可以符合Error，不过枚举是迄今为止最常见的Error。（点击Xcode工具栏中的后退箭头回到playground。）

现在有了可以抛出的类型，就能在lex()方法中实现default分支来抛出Error枚举的实例了，如代码清单20-10所示。

代码清单20-10　抛出错误

```
...
class Lexer {
    ...
    func lex() throws -> [Token] {
        var tokens = [Token]()

        while let nextCharacter = peek() {
            switch nextCharacter {
            case "0" ... "9":
                // 开始处理数，需要获取数的剩余部分

            case "+":
                tokens.append(.plus)
                advance()

            case " ":
                // 前进，忽略空格
                advance()

            default:
                // 出乎意料——需要返回错误
                throw Lexer.Error.invalidCharacter(nextCharacter)
            }
        }

        return tokens
    }
}
```

跟return一样，throw会让函数马上停止运行，并回到其调用者。

最后，词法分析器需要从输入中取出整数。创建方法getNumber()，它会用lex()用到的peek()和advance()一个数字一个数字地进行组合。

接着更新lex()，添加对getNumber()的调用，把数拼接到符号数组中，如代码清单20-11所示。

代码清单20-11　实现Lexer.getNumber()

```
...
class Lexer {
    ...
    func getNumber() -> Int {
        var value = 0

        while let nextCharacter = peek() {
            switch nextCharacter {
            case "0" ... "9":
                // 还有数字，加到结果上
                let digitValue = Int(String(nextCharacter))!
                value = 10*value + digitValue
                advance()
```

```
            default:
                // 非数字，回到常规的词法分析
                return value
            }
        }

        return value
    }

    func lex() throws -> [Token] {
        var tokens = [Token]()

        while let nextCharacter = peek() {
            switch nextCharacter {
            case "0" ... "9":
                // 开始处理数，需要获取数的剩余部分
                let value = getNumber()
                tokens.append(.number(value))

            case "+":
                tokens.append(.plus)
                advance()

            case " ":
                // 前进，忽略空格
                advance()

            default:
                throw Lexer.Error.invalidCharacter(nextCharacter)
            }
        }
        return tokens
    }
}
```

到了这里，所有的编译错误应该都消失了。

getNumber()会循环处理输入字符，把数字累加到一个整数值上。注意，这里在Int(String(nextCharacter))!中用到了之前提醒过要慎用的功能——强制展开可空实例。不过，这种情况下是绝对安全的。因为我们知道nextCharacter有一个数字，把它转化为整型一定会成功，不可能返回nil。只要getNumber()遇到非数字字符（或者输入结束），它就会停止并返回累加的值。

Lexer完成了。现在该测试一下了。写一个新函数，让它接受一个输入字符串并进行词法分析，并用一些测试输入调用这个函数，如代码清单20-12所示。（这个函数还不能正常工作——在敲代码的时候想想为什么。）

代码清单20-12　运行词法分析器

```
...
func evaluate(_ input: String) {
```

```
        print("Evaluating: \(input)")
        let lexer = Lexer(input: input)
        let tokens = lexer.lex()
        print("Lexer output: \(tokens)")
    }

    evaluate("10 + 3 + 5")
    evaluate("1 + 2 + abcdefg")
```

evaluate(_:)接受一个输入字符串，创建一个Lexer，并把输入解析为Token；但是编译器不接受这段代码。注意调用lex()那一行的错误信息：Call can throw, but it is not marked with 'try' and the error is not handled。编译器是在告诉你，因为lex()方法被标记为throws，所以调用lex()一定要准备好处理错误。

20.3 捕获错误

Swift使用一种我们还没有见过的控制结构来处理错误：do/catch。在do中至少有一个try语句，我们稍后会解释。首先，修改evaluate(_:)，用这个控制流处理来自lex()的错误，如代码清单20-13所示。

代码清单20-13 evaluate(_:)中的错误处理

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)

    do {
        let tokens = try lexer.lex()
        print("Lexer output: \(tokens)")
    } catch {
        print("An error occurred: \(error)")
    }
}
```

这些新的关键字都是什么意思？do引入了新的作用域，跟if语句很像。在do的作用域内，可以像平时那样写代码，比如调用print()。除此之外，还可以调用被throws标记的函数或方法。每个这样的调用都必须用try关键字标记。

在do语句块的最后要写一个catch语句块。如果do语句块中的任意一个try调用抛出错误，catch语句块就会运行，并且会把抛出的错误绑定到常量error上。

现在应该能在调试区域看到evaluate(_:)运行后的输出了。

```
Evaluating: 10 + 3 + 5
Lexer output: [Token.number(10), Token.plus,
               Token.number(3), Token.plus, Token.number(5)]
Evaluating: 1 + 2 + abcdefg
An error occurred: Lexer.Error.invalidCharacter("a")
```

上面的catch语句块没有指定某类错误，所以会捕获所有的Error。可以再添加一个catch语句块捕获某类错误。在本例中，我们知道词法分析器会抛出Lexer.Error.invalidCharacter错误，所以加一个catch语句块捕获它，如代码清单20-14所示。

代码清单20-14　捕获invalidCharacter错误

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)

    do {
        let tokens = try lexer.lex()
        print("Lexer output: \(tokens)")
    } catch Lexer.Error.invalidCharacter(let character) {
        print("Input contained an invalid character: \(character)")
    } catch {
        print("An error occurred: \(error)")
    }
}
```

添加的catch语句块是专门寻找Lexer.Error.invalidCharacter错误的。catch语句块支持模式匹配，就跟switch语句一样，因此可以把无效字符绑定到常量里并且在catch语句块中使用。现在应该能看到更明确的错误信息了。

```
Evaluating: 10 + 3 + 5
Lexer output: [Token.number(10), Token.plus,
               Token.number(3), Token.plus, Token.number(5)]
Evaluating: 1 + 2 + abcdefg
Input contained an invalid character: a
```

恭喜，编译器的词法分析阶段完成了！在进入下一阶段之前，删除会导致错误的evaluate(_:)调用，如代码清单20-15所示。

代码清单20-15　删除坏的输入

```
...
evaluate("10 + 3 + 5")
evaluate("1 + 2 + abcdefg")
```

20.4　解析符号数组

既然完成了词法分析器，那就可以把输入字符串转化为符号数组了；每个符号不是.number就是.plus。下一步是写一个解析器，它的作用是计算从词法分析器传过来的符号序列。举个例子，给解析器传入[.number(5), .plus, .number(3)]应该会得到答案8。解析这个符号序列的算法比词法分析用的算法有更多限制，因为符号出现的顺序很重要。规则是这样的：

- 第一个符号必须是数；
- 解析完一个数后，要么解析器已经到达输入终点，要么下一个符号必须是.plus；

- 解析完.plus后，下一个符号必须是数。

解析器写起来跟词法分析器差不多，不过还要简单点。解析器不需要区分peek()方法和advance()方法，两者可以合并为getNextToken()方法，这个方法会返回下一个Token，如果所有的符号都用完了就返回nil。

创建有getNextToken()方法的Parser类，如代码清单20-16所示。

代码清单20-16 开始实现Parser

```
...
class Lexer {
    ...
}

class Parser {
    let tokens: [Token]
    var position = 0

    init(tokens: [Token]) {
        self.tokens = tokens
    }

    func getNextToken() -> Token? {
        guard position < tokens.count else {
            return nil
        }
        let token = tokens[position]
        position += 1
        return token
    }
}

func evaluate(_ input: String) {
    ...
}
...
```

Parser用符号数组初始化，开始的position是0。getNextToken()方法用guard来检查是否有剩下的符号；如果有，就返回下一个，并把position移动到它返回的符号后面。

解析器的三条规则中有两条提到了“必须是数”。实现解析器的不错起点是写一个获取数的方法。如果下一个符号必须是数，那就要考虑两种错误的情形。解析器可能已经到达了符号数组的末尾，也就是没有数了。也有可能下一个符号是.plus而不是数。举个例子，有人可能会给解析器这样的输入字符串"10 + + 5"。

定义一个符合Error的错误枚举表示这两种情形，如代码清单20-17所示。

代码清单20-17 定义可能的Parser错误

```
...
class Parser {
    enum Error: Swift.Error {
```

```
        case unexpectedEndOfInput
        case invalidToken(Token)
    }

    let tokens: [Token]
    var position = 0
    ...
}
...
```

既然现在能表达在获取数过程中可能遇到的错误了，那就可以添加一个方法用来获取下一个.number符号的值；如果无法获取的话就抛出错误，如代码清单20-18所示。

代码清单20-18　实现Parser.getNumber()

```
...
class Parser {
    ...

    func getNextToken() -> Token? {
        guard position < tokens.count else {
            return nil
        }
        let token = tokens[position]
        position += 1
        return token
    }

    func getNumber() throws -> Int {
        guard let token = getNextToken() else {
            throw Parser.Error.unexpectedEndOfInput
        }

        switch token {
        case .number(let value):
            return value
        case .plus:
            throw Parser.Error.invalidToken(token)
        }
    }
}
...
```

getNumber()方法的签名是throws -> Int，所以这是一个正常情况下会返回整型但是也可能会抛出错误的函数。用guard语句来检查是否至少还有一个符号。注意在guard的else语句块中，可以用throw代替return，因为guard只要求else语句块让函数停止执行并返回调用者。在确认有符号后，用switch语句取出数值（如果符号是.number）或者抛出invalidToken错误（如果是.plus）。

有了getNumber()，实现解析算法的剩余部分就挺简单了。添加parse()方法完成这个任务，如代码清单20-19所示。

代码清单20-19　实现Parser.parse()

```
...
class Parser {
    ...
    func parse() throws -> Int {
        // 第一个应该是数
        var value = try getNumber()

        while let token = getNextToken() {
            switch token {

            // 数后面为加号是合法的
            case .plus:
                // 加号后面必须又是一个数
                let nextNumber = try getNumber()
                value += nextNumber

            // 数后面还是数是不合法的
            case .number:
                throw Parser.Error.invalidToken(token)
            }
        }

        return value
    }
}
...
```


parse()的实现跟上面概述的解析算法匹配。输入必须从数开始（value的初始化）。解析到一个数后，遍历剩余的符号。如果下一个符号是.plus，那再下一个符号就必须是.number。当到达符号的终点时，while循环结束，然后返回value。

这里有些新的东西。我们使用try关键字标记了getNumber()的调用。这是Swift要求的，因为getNumber()可能抛出错误。不过，没有用do/catch语句块。为什么这里Swift允许在没有do语句块的情况下用try？

Swift要求用try标记的方法调用"处理错误"。很容易把"处理错误"理解为捕获错误，就像evaluate(_:)里那样。但是还有一种合理的方式处理错误：再次抛出！这就是本例中的情况。因为parse()本身是一个可能抛出错误的方法，所以可以在其内部没有do/catch的情况下调用try。如果某一个有try的调用失败了，错误会被"再次抛出"到parse()外面。

现在解析器完成了。更新evaluate(_:)来调用解析器，并处理Parser可能抛出的某些错误，如代码清单20-20所示。

代码清单20-20　更新evaluate(_:)使用Parser

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)
```

```
        do {
            let tokens = try lexer.lex()
            print("Lexer output: \(tokens)")

            let parser = Parser(tokens: tokens)
            let result = try parser.parse()
            print("Parser output: \(result)")
        } catch Lexer.Error.invalidCharacter(let character) {
            print("Input contained an invalid character: \(character)")
        } catch Parser.Error.unexpectedEndOfInput {
            print("Unexpected end of input during parsing")
        } catch Parser.Error.invalidToken(let token) {
            print("Invalid token during parsing: \(token)")
        } catch {
            print("An error occurred: \(error)")
        }
    }
    ...
```

现在应该能看到两步编译器可以成功计算输入的表达式了：

```
Evaluating: 10 + 3 + 5
Lexer output: [Token.number(10), Token.plus,
               Token.number(3), Token.plus, Token.number(5)]
Parser output: 18
```

试着改变输入，增加或减少数。尝试某些能通过词法分析（也就是只包含合法符号）但是会造成解析器抛出错误的输入。两个简单的例子是"10 + 3 5"和"10 + "。

20.5　用鸵鸟政策处理错误

你已经看到必须用try标记每次调用可能抛出错误的函数，而任何用try标记的调用必须要么在do/catch语句块内，要么在一个本身被标记为throws的函数内。这两条规则配合起来能确保我们处理任何潜在的错误。试着修改evaluate(_:)函数来破坏其中一条规则，如代码清单20-21所示。

代码清单20-21　把evaluate(_:)改得不合法

```
    ...
    func evaluate(_ input: String) {
        print("Evaluating: \(input)")
        let lexer = Lexer(input: input)
        let tokens = try lexer.lex()

        do {
            let tokens = try lexer.lex()
            print("Lexer output: \(tokens)")

            let parser = Parser(tokens: tokens)
            let result = try parser.parse()
            print("Parser output: \(result)")
```

```
        } catch Lexer.Error.invalidCharacter(let character) {
            print("Input contained an invalid character: \(character)")
        } catch Parser.Error.unexpectedEndOfInput {
            print("Unexpected end of input during parsing")
        } catch Parser.Error.invalidToken(let token) {
            print("Invalid token during parsing: \(token)")
        } catch {
            print("An error occurred: \(error)")
        }
    }
    ...
```

我们把`try lexer.lex()`移到do语句块外面，所以现在编译器报错了。编译错误是：`Errors thrown from here are not handled`。可以告诉Swift编译器我们不想处理潜在的错误。把`try`改成`try!`来看看效果，如代码清单20-22所示。

代码清单20-22 在`evaluate(_:)`中使用`try!`

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)
    let tokens = trytry! lexer.lex()

    ...
}
...
```

现在代码可以编译了，但是要小心。如果`lexer.lex()`抛出错误，Swift会怎么做？`try!`关键字末尾的感叹号是个很强的暗示。与强制展开可空实例一样，如果用强制性的关键字`try!`，则一旦出现错误程序就会触发陷阱。

之前有一个`evaluate(_:)`调用会导致词法分析器抛出错误。把这个调用添加回去看看会发生什么，如代码清单20-23所示。

代码清单20-23 用`try!`对坏的输入进行词法分析

```
...
evaluate("10 + 3 + 5")
evaluate("1 + 2 + abcdefg")
```

我们看不到无效符号的错误信息，程序直接在`try! lexer.lex()`这一行触发了陷阱。

我们之前建议过避免使用强制展开可空实例和隐式展开可空实例。我们更加强烈建议避免使用`try!`。只有当程序无法处理错误而你也确实想让程序在发生错误时触发陷阱（或者崩溃，如果程序在用户的设备上运行）的时候才能用`try!`。

`try`还有第三种变体，可以在发生错误时忽略错误而不触发陷阱。可以用`try?`调用一个可能抛出错误的函数，得到函数原本的返回值对应的可空类型返回值。这意味着需要类似`guard`这样的工具检查可空实例是否有值。

把会触发陷阱的`try!`改成`guard`和`try?`的组合，如代码清单20-24所示。

代码清单20-24　在evaluate(_:)中使用try?

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)
    let tokens = try! lexer.lex()
    guard let tokens = try? lexer.lex() else {
        print("Lexing failed, but I don't know why")
        return
    }
    ...
}
...
```

try?没有try!那么邪恶，但是我们仍然建议大多数时候避免使用。在用try?调用函数时，必须处理返回值是nil的可能性。上面的代码对evaluate(_:)的返回值使用了guard。不过，用catch处理错误通常更好，因为可以拿到函数抛出的错误。

try?最有用的情况是，在被调用的函数可能失败时还有一个后备函数可用。不过evaluate(_:)没有这样的后备函数，所以还是回到之前的错误处理机制，如代码清单20-25所示。

代码清单20-25　恢复evaluate(_:)

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)
    guard let tokens = try? lexer.lex() else {
        print("Lexing failed, but I don't know why")
        return
    }

    do {
        let tokens = try lexer.lex()
        print("Lexer output: \(tokens)")

        let parser = Parser(tokens: tokens)
        let result = try parser.parse()
        print("Parser output: \(result)")
    } catch Lexer.Error.invalidCharacter(let character) {
        print("Input contained an invalid character: \(character)")
    } catch Parser.Error.unexpectedEndOfInput {
        print("Unexpected end of input during parsing")
    } catch Parser.Error.invalidToken(let token) {
        print("Invalid token during parsing: \(token)")
    } catch {
        print("An error occurred: \(error)")
    }
}
...
```

20.6　Swift 的错误处理哲学

Swift的设计理念是鼓励写安全、易读的代码，它的错误处理系统也一样。任何可能失败的函数都应该用`throws`标记。这样从函数类型就能明显看出是否需要处理潜在的错误。

Swift还需要我们把所有可能失败的函数调用用`try`标记。这对Swift代码的阅读者来说极为有利。如果一个函数调用被标记为`try`，你就知道它是潜在的问题源，必须处理。如果函数没有用`try`标记，那你就知道它永远不会抛出需要处理的错误。

如果你用过C++或Java，那么很重要的一点是要注意区分Swift的错误处理和基于异常的错误处理的区别。虽然Swift用了某些相同的术语，尤其是`try`、`catch`和`throw`，但是Swift不是用异常来实现错误处理的。在把一个函数标记为`throws`时，其实是把返回值类型从正常的类型变为“要么是正常返回的类型，要么是`Error`协议的实例”。

最后，还有一个深深植入Swift的重要的错误处理哲学。带`throws`的函数不会声明自己会抛出什么样的错误。这样会产生两个实际的影响。第一，不需要修改函数的API就可以随意添加潜在的`Error`。第二，在用`catch`处理错误时，必须总是准备好处理未知的错误类型。

编译器会强制保证第二点。试着修改`evaluate(_:)`，把最后一个`catch`语句块删掉，如代码清单20-26所示。

代码清单20-26　在`evaluate(_:)`中回避未知的`Error`处理

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)

    do {
        let tokens = try lexer.lex()
        print("Lexer output: \(tokens)")

        let parser = Parser(tokens: tokens)
        let result = try parser.parse()
        print("Parser output: \(result)")
    } catch Lexer.Error.invalidCharacter(let character) {
        print("Input contained an invalid character: \(character)")
    } catch Parser.Error.unexpectedEndOfInput {
        print("Unexpected end of input during parsing")
    } catch Parser.Error.invalidToken(let token) {
        print("Invalid token during parsing: \(token)")
    } catch {
        print("An error occurred: \(error)")
    }
}
...
```

现在编译器会在`do`语句块中两行有`try`的函数调用的代码上提示错误。错误消息看起来很熟悉：`Errors thrown from here are not handled because the enclosing catch is not exhaustive`。与`switch`语句一样，Swift会对`do/catch`语句块进行全覆盖检查，必须确保处理

所有潜在的Error。

恢复会处理任何类型错误的catch语句块来修复evaluate(_:)的错误，如代码清单20-27所示。

代码清单20-27　evaluate(_:)中错误处理的全覆盖

```
...
func evaluate(_ input: String) {
    print("Evaluating: \(input)")
    let lexer = Lexer(input: input)

    do {
        let tokens = try lexer.lex()
        print("Lexer output: \(tokens)")

        let parser = Parser(tokens: tokens)
        let result = try parser.parse()
        print("Parser output: \(result)")
    } catch Lexer.Error.invalidCharacter(let character) {
        print("Input contained an invalid character: \(character)")
    } catch Parser.Error.unexpectedEndOfInput {
        print("Unexpected end of input during parsing")
    } catch Parser.Error.invalidToken(let token) {
        print("Invalid token during parsing: \(token)")
    } catch {
        print("An error occurred: \(error)")
    }
}
...
```

20.7　青铜挑战练习

现在表达式求值器只支持加法，这样不是很有用！添加对减法的支持。应该能够调用evaluate("10 + 5 - 3 - 1")并得到输出11。

20.8　白银挑战练习

evaluate(_:)打印的错误信息有用，但是还有改进的空间。下面是一些错误输入和产生的错误信息：

```
evaluate("1 + 3 + 7a + 8")
> Input contained an invalid character: a

evaluate("10 + 3 3 + 7")
> Invalid token during parsing: .number(3)
```

通过引入错误发生所在的字符位置来让信息更有用。完成这个挑战后，错误信息看起来应该是这样的：

```
evaluate("1 + 3 + 7a + 8")
> Input contained an invalid character at index 9: a

evaluate("10 + 3 3 + 7")
> Invalid token during parsing at index 7: 3
```

提示：你需要结合错误位置和已有的错误类型。要把`String.CharacterView.Index`转化成一个位置数，可以对字符视图调用`distance(from:to:)`方法。下面举个例子，如果`input`是`String.CharacterView`，而`position`是`String.CharacterView.Index`，下面这行代码会计算字符串的开头和`position`之间有多少字符：

```
let distanceToPosition = input.distance(from: input.startIndex, to: position)
```

20.9 黄金挑战练习

现在再加把劲，给计算器增加乘法和除法的支持。如果觉得这跟增加减法一样简单，那就再想想！求值器应该给乘除法比加减法更高的优先级。下面是一些示例输入和期望的输出。

```
evaluate("10 * 3 + 5 * 3") // 应该打印45
evaluate("10 + 3 * 5 + 3") // 应该打印28
evaluate("10 + 3 * 5 * 3") // 应该打印55
```

如果没有头绪了，那就搜索“递归下降解析器”，这是我们一直在实现的解析器。有个提示能帮助你起步：不要解析一个数然后预期下一个是`.plus`或`.minus`，而是解析由数和乘除号计算出的项，然后预期下一个是`.plus`或`.minus`。

第21章 扩 展

想象一下，你正在开发的应用会特别频繁地使用Swift标准库中的某个类型——假设是双精度浮点数`Double`。如果`Double`类型能基于你在应用中的使用方式支持某些额外的方法，那么开发过程会变得更容易。不幸的是，你没有`Double`的实现代码，所以无法自己直接添加功能。那该怎么办呢?

Swift提供一个叫*扩展*（extension）的特性，该特性就是为这种情况设计的。扩展能让你给已有的类型添加功能，可以用来扩展结构体、枚举和类。

对类型的扩展支持以下几种能力：

- 添加计算属性；
- 添加新初始化方法；
- 使类型符合协议；
- 添加新方法；
- 添加嵌入类型。

本章会利用扩展为缺少定义和实现细节的已有类型添加功能，还会利用扩展为自定义类型添加功能。这两种情况都会以模块化的方式为类型添加功能，也就是把相似功能放在一个扩展中。

21.1 扩展已有类型

新建一个playground，命名为Extensions。我们会在这个playground中为汽车的行为建模。

速度是机动车的重要特征。因为速度可能有小数值，所以用双精度浮点数表示比较合理。

既然要频繁使用双精度浮点数，那么用一种与背景相关的方式引用它会比较有用。Swift的`typealias`关键字提供了一种给已有类型起别名的方式。给`Double`类型起个别名，如代码清单21-1所示。

代码清单21-1 建立类型别名

```
import Cocoa

var str = "Hello, playground"

typealias Velocity = Double
```

typealias关键字可以把Velocity定义为Double的别名。这种可交换性可以让名字跟使用场景更加相关，从而有助于在本章要写的扩展中场景化双精度浮点数。

设置好typealias后，就可以扩展类型，让它支持速度常见单位之间的换算。Swift扩展不允许为类型添加存储属性。我们将利用扩展添加两个计算属性，如代码清单21-2所示。

代码清单21-2　扩展Velocity，让它支持mph和kph

```
...
typealias Velocity = Double
extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
```

extension关键字表示要扩展Velocity。我们给Velocity添加了两个计算属性：kph和mph。Velocity的这两个属性分别表示机动车用每小时公里数（kph）和每小时英里数（mph）表示的速度。注意，这个扩展把mph作为默认单位：这个计算属性只是简单返回self；而kph则要做换算。

虽然typealias提供的可交换性带来了一定的好处，但是也有点复杂。在这个文件中可能会遇到这么一种情况：你想用Double而不是Velocity。因为Velocity和Double是可交换的，所以在Velocity上定义的扩展对Double也有效。通过类型别名Velocity给Double添加扩展能给出有帮助的上下文信息，说明Double也能用这两个计算属性，但是它们只有和Velocity类型别名一起用时才有意义。

回忆一下，本章的一个目标就是定义机动车的行为。协议是Swift对于类型定义接口最有用的特性。可以用扩展让类型符合协议。

新建协议Vehicle描述机动车的基本特征，如代码清单21-3所示。

代码清单21-3　用扩展让类型符合协议

```
...
typealias Velocity = Double
extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
protocol Vehicle {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}
```

Vehicle声明了三个属性：topSpeed、numberOfDoors和hasFlatbed。每个属性只需要让符合这个协议的类型实现该属性的读取方法即可。符合这个协议的类型需要提供这些描述机动车通用特征的属性。

21.2　扩展自己的类型

现在需要先新建一个类型，然后利用扩展让它符合协议。新建结构体Car，如代码清单21-4所示。稍后会用Car的扩展让它符合Vehicle协议。

代码清单21-4　结构体Car

```
...
typealias Velocity = Double
extension Velocity {
    var kph: Velocity { return self * 1.60934 }
    var mph: Velocity { return self }
}
protocol Vehicle {
    var topSpeed: Velocity { get }
    var numberOfDoors: Int { get }
    var hasFlatbed: Bool { get }
}
struct Car {
    let make: String
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue >= 0.0,
                         "New value must be between 0 and 1.")
        }
    }
}
```

这里定义了新结构体Car。Car定义了一组实例存储属性。除了gasLevel，其他属性都是常量。

gasLevel是可变存储属性，还带有属性观察者。我们每次为gasLevel设置新值之前，willSet观察者都会被调用，在其实现内部用precondition()确保赋给gasLavel的newValue介于0和1之间。这些值用百分比的形式表示油箱有多满。

21.2.1　用扩展使类型符合协议

扩展提供了很好的机制支持对相关的功能分组。把相关功能放进一个扩展有助于代码的可读性和可维护性。这种模式也有助于类型的接口保持整洁。

扩展Car使其符合VehicleType协议，如代码清单21-5所示。

代码清单21-5　扩展Car使其符合VehicleType

```
...
struct Car {
    let make: String
```

```
    let model: String
    let year: Int
    let color: String
    let nickname: String
    var gasLevel: Double {
        willSet {
            precondition(newValue <= 1.0 && newValue >= 0.0,
                         "New value must be between 0 and 1.")
        }
    }
}
extension Car: Vehicle {
    var topSpeed: Velocity { return 180 }
    var numberOfDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
```

这个新扩展使Car符合Vehicle。符合协议的语法和前面出现过的一样，但是这次是用扩展完成的：extension Car: Vehicle。

我们在扩展体内实现协议所需的属性，给每个属性写了一个简单的读取方法。为简单起见，协议的每个属性都返回默认值。

21.2.2　用扩展添加初始化方法

前面提到过，如果结构体没有初始化方法的话，会自带一个成员初始化方法。如果想给结构体写一个新的初始化方法，同时又不想失去成员初始化方法，那就可以用扩展给结构体添加初始化方法。在扩展中为Car添加一个初始化方法，如代码清单21-6所示。

代码清单21-6　扩展Car，添加初始化方法

```
...
extension Car: Vehicle {
    var topSpeed: Velocity { return 180 }
    var numberOfDoors: Int { return 4 }
    var hasFlatbed: Bool { return false }
}
extension Car {
    init(make: String, model: String, year: Int) {
        self.init(make: make,
            model: model,
            year: year,
            color: "Black",
            nickname: "N/A",
            gasLevel: 1.0)
    }
}
```

Car的新扩展添加了一个初始化方法。这个初始化方法的参数对应实例的make、model和year属性。新初始化方法的参数会被传递给Car自带的成员初始化方法，我们还为缺失的参数提供了默认值。这两个初始化方法一起确保了Car的实例为所有的属性赋值。

Car的成员初始化方法之所以能保留下来，是因为新初始化方法是在扩展中定义和实现的。这个模式很有用。

新建一个Car实例，看一下怎么使用扩展中定义的初始化方法，如代码清单21-7所示。

代码清单21-7　Car实例

```
...
extension Car {
    init(make: String, model: String, year: Int) {
        self.init(make: make,
            model: model,
            year: year,
            color: "Black",
            nickname: "N/A",
            gasLevel: 1.0)
    }
}
var c = Car(make: "Ford", model: "Fusion", year: 2013)
```

上面的代码新建了一个实例c。这个实例是用Car的扩展中定义的一个初始化方法创建的。看一下运行结果侧边栏，你会看到c的属性的值就是我们给新初始化方法传递的，应该也能看到给成员初始化方法传递的默认值。

21.2.3　嵌套类型和扩展

Swift的扩展还能给已有的类型添加嵌套类型。举个例子，假设你要给Car添加一个枚举来协助汽车的分类。在Car上新建扩展，添加嵌套类型，如代码清单21-8所示。

代码清单21-8　创建带嵌套类型的扩展

```
...
var c = Car(make: "Ford", model: "Fusion", year: 2013)
extension Car {
    enum Kind {
        case coupe, sedan
    }
    var kind: Kind {
        if numberOfDoors == 2 {
            return .coupe
        } else {
            return .sedan
        }
    }
}
```

Car的这个新扩展添加了嵌套类型Kind。Kind是有两个成员的枚举：一个是coupe，另一个是sedan。扩展还为Car实例添加了一个计算属性kind，表示汽车种类。嵌套类型也符合CustomStringConvertible，以方便打印信息。

kind根据汽车的门数返回嵌套枚举的值：如果是两门，就是轿跑车（coupe）；否则就是三厢轿车（sedan）。

访问前面创建的计算属性kind，练习使用扩展中的嵌套类型，如代码清单21-9所示。

代码清单21-9　访问kind

```
...
extension Car {
    enum Kind {
        case coupe, sedan
    }
    var kind: Kind {
        if numberOfDoors == 2 {
            return .coupe
        } else {
            return .sedan
        }
    }
}
c.kind
```

运行结果侧边栏应该会显示sedan。

21.2.4　扩展中的函数

可以用扩展给已有类型添加函数。举个例子，你可能已经注意到Car没有加油函数。创建扩展为Car添加这个功能，如代码清单21-10所示。

代码清单21-10　用扩展添加函数

```
...
c.kind
extension Car {
    mutating func emptyGas(by amount: Double) {
        precondition(amount <= 1 && amount > 0,
                     "Amount to remove must be between 0 and 1.")
        gasLevel -= amount
    }

    mutating func fillGas() {
        gasLevel = 1.0
    }
}
```

这个新扩展为Car类型添加了两个函数：emptyGas(by:)和fillGas()。注意，两个函数都用mutating标记了。为什么？记住，Car是结构体。如果函数要改变结构体的属性的值，就必须用mutating关键字声明。

emptyGas(by:)函数接受一个参数：从油箱放出的油量。在emptyGas(by:)函数内使用precondition()来确保从油箱中放出的油应该介于0到1之间。fillGas()的实现只是简单地把gasLevel属性设置为满，即1.0。

在汽车上练习使用新函数，如代码清单21-11所示。

代码清单21-11　放油，再加满油

```
...
extension Car {
    mutating func emptyGas(by amount: Double) {
        precondition(amount <= 1 && amount > 0,
                     "Amount to remove must be between 0 and 1.")
        gasLevel -= amount
    }

    mutating func fillGas() {
        gasLevel = 1.0
    }
}
c.emptyGas(by: 0.3)
c.gasLevel
c.fillGas()
c.gasLevel
```

用了emptyGas(by:)后，你应该能看到运行结果侧边栏中显示油量是0.7。加满油后，油量是1.0。

21.3　青铜挑战练习

扩展Int类型，添加一个timesFive计算属性。这个计算属性应该返回该整数乘以5的结果。这个属性应该这么用：

```
5.timesFive // 25
```

21.4　青铜挑战练习

有时候，一开始写出来的代码看上去没问题，但是实际使用的时候发现不太对。用扩展让Car符合Vehicle时也是这样。

在让Car符合Vehicle协议时，我们添加了总是返回4的numberOfDoors计算属性。这其实是把numberOfDoors变成了Car的常量。结果就是，kind里的条件语句总是返回.sedan。由于Car符合Vehicle的实现方式，不会有其他可能的值了。

重构Car，使它有一个常量存储属性numberOfDoors。注意，这个改变意味着还要做别的修改。利用编译器的报错指导你的解决方案。

21.5　白银挑战练习

emptyGas(by:)方法有些bug。举个例子，如果当前的gasLevel小于要放掉的油量，属性的新值会是负数。负数不合理，而且会让程序停止运行（还记得gasLevel属性观察者里的precondition()吧）。修改emptyGas(by:)的实现，确保gasLevel()不会减少到变成负数。

第 22 章 泛 型

22

到目前为止，我们写过的所有性和函数都是在具体的类型上工作的，比如Int、String和Monster。你可能已经注意到了，Swift允许数组包含任何类型。你可以创建内建Swift类型的数组，比如[Int]和[Double]，还有自定义类型的数组，比如[Monster]和[Person]。数组是怎么实现的？怎么能用一种方式写出对很多类型都适用的代码呢？这两个问题的答案都是“泛型”。

Swift泛型（generics）让我们写出的类型和函数可以使用对于我们或编译器都未知的类型。本书中用到的很多内建类型（包括可空类型、数组和字典）都是用泛型实现的。在本章中，我们会研究如何写泛型类型（跟数组很像）。你还会学到如何用泛型写出灵活的函数以及泛型和协议之间有怎样的关系。

22.1　泛型数据结构

下面要创建一个泛型栈（stack），这是计算机科学中历史悠久的数据结构。栈是后进先出的数据结构。它支持两种基本操作，一种是把数据推（push）进栈，一种是把最近推进栈的数据弹（pop）出栈。

首先新建playground，命名为Generics。然后创建一个只保存整数的Stack结构体，如代码清单22-1所示。

代码清单22-1　创建Stack

```
import Cocoa

var str = "Hello, playground"

struct Stack {
    var items = [Int]()

    mutating func push(_ newItem: Int) {
        items.append(newItem)
    }

    mutating func pop() -> Int? {
        guard !items.isEmpty else {
            return nil
```

```
        }
        return items.removeLast()
    }
}
```

这个结构体有三个值得一提的地方。第一，存储属性items是保存当前栈中数据的数组。第二，push(_:)方法把新数据项放到items数组的末尾，从而把它推进栈。第三，pop()方法调用数组的removeLast()方法把栈顶数据项弹出来。它会同时删除最后一个数据项并且将其返回。注意pop()返回可空整型，因为栈可能是空的（这种情况下没有东西可以弹出）。

创建一个Stack实例看一下实际使用，如代码清单22-2所示。

代码清单22-2　创建Stack实例

```
...
var intStack = Stack()
intStack.push(1)
intStack.push(2)

print(intStack.pop()) // 打印Optional(2)
print(intStack.pop()) // 打印Optional(1)
print(intStack.pop()) // 打印nil
```

这段代码新建了一个Stack实例，推进去两个值，然后尝试弹出三个值。如我们所料，调用pop()会以相反的顺序返回之前推进去的整数，而且当栈中没有数据项时，pop()会返回nil。

Stack存储Int很好用，但是目前仅限于Int。如果Stack能变得更通用就好了。修改Stack，使它变成一个泛型数据结构，可以保存任何类型而不只是Int，如代码清单22-3所示。

代码清单22-3　把Stack变成泛型

```
...
struct Stack<Element> {
    var items = [IntElement]()

    mutating func push(_ newItem: IntElement) {
        items.append(newItem)
    }

    mutating func pop() -> Int?Element? {
        guard !items.isEmpty else {
            return nil
        }
        return items.removeLast()
    }
}
...
```

这段代码给Stack的声明添加了一个名为Element的占位类型（placeholder type）。Swift声明泛型的语法是在类型名后面紧跟尖括号（<>）。尖括号内的名字表示占位类型：<Element>。占位类型Element可以在Stack结构体内任何一个能用具体类型的地方使用。可以把这种用法看作把所有出现Int的地方换成Element，包括属性声明、push(_:)中的参数类型以及pop()的返回值类型。

现在创建Stack实例的地方有一个编译错误，因为没有指定应该用什么类型替换占位类型Element。编译器用实际类型替换占位类型的过程被称为特化（specialization）。特化的完整细节超出了本书的范围，不过简单来说，就是指它能让编译器把应用变得更快，因为编译器能产生知道实际使用的类型是什么的代码。指定intStack是Stack特化为整型的实例可以修复这个错误。我们用尖括号语法实现这一点，如代码清单22-4所示。

代码清单22-4 特化intStack

```
...
var intStack = Stack<Int>()
...
```

这样就解决了编译错误。

现在可以创建任何类型的Stack了。创建一个字符串的Stack，如代码清单22-5所示。

代码清单22-5 创建一个字符串的Stack

```
...
print(intStack.pop()) // 打印Optional(1)
print(intStack.pop()) // 打印nil

var stringStack = Stack<String>()
stringStack.push("this is a string")
stringStack.push("another string")

print(stringStack.pop()) // 打印Optional("another string")
```

22

注意，虽然intStack和StringStack都是Stack的实例，但是它们的类型不同，这一点很重要。intStack的类型是Stack<Int>，给inStack.push(_:)传递除了整型以外的类型会产生编译错误。与其类似，stringStack的类型是Stack<String>，跟Stack<Int>不同。

泛型数据结构不仅常见，而且很有用。用和结构体一样的语法也可以把类和枚举变为泛型类型。此外，在Swift中，不仅类型可以是泛型，函数和方法也可以。

22.2 泛型函数和方法

还记得第13章中数组的map(_:)方法吗？map(_:)会对数组中的每个元素调用闭包并返回结果的数组。学习了泛型之后，就可以自己实现这个函数了。把代码清单22-6所示的代码添加到playground中。

代码清单22-6 你自己的map函数

```
...
func myMap<T,U>(_ items: [T], _ f: (T) -> (U)) -> [U] {
    var result = [U]()
    for item in items {
        result.append(f(item))
    }
```

```
        return result
    }
```

如果没有见过其他语言的泛型，你可能会觉得myMap(_:_:)的声明看起来很丑。声明中没有我们熟悉的具体类型，取而代之的是T和U，而且符号和标点比字母还多！唯一的新变化只是从一个占位类型变成了两个。图22-1显示了对这行代码的分解说明。

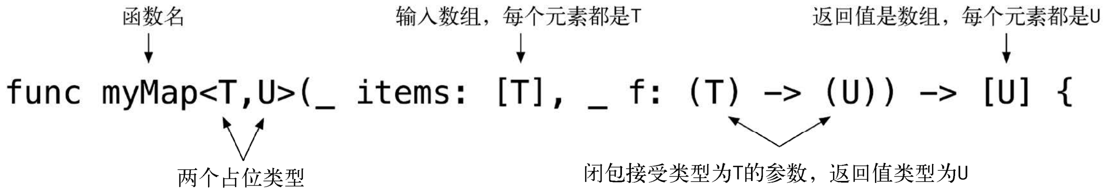


图22-1　myMap声明

myMap(_:_:)的使用方式跟map(_:)一样。创建一个字符串数组，然后把它变换成字符视图长度的数组，如代码清单22-7所示。

代码清单22-7　变换数组

```
...
func myMap<T,U>(_ items: [T], _ f: (T) -> (U)) -> [U] {
    ...
}

let strings = ["one", "two", "three"]
let stringLengths = myMap(strings) { $0.characters.count }
print(stringLengths) // 打印[3, 3, 5]
```

传递给myMap(_:_:)的闭包必须接受一个参数，其类型和items数组中的元素类型一致，但是返回值可以是任何类型。在本例对myMap(_:_:)的调用中，用字符串替换T，用整型替换U。（注意，在实际项目中没有必要声明自己的变换函数——用内建的map(_:)就行。）

方法也可以用泛型，即使是在本身已经是泛型的类型内部也可以。刚才的myMap(_:_:)函数只对数组有效，但是变换Stack的需求似乎也挺合理。在Stack上创建一个map(_:)方法，如代码清单22-8所示。

代码清单22-8　变换Stack

```
...
struct Stack<Element> {
    var items = [Element]()

    mutating func push(_ newItem: Element) {
        items.append(newItem)
    }

    mutating func pop() -> Element? {
        guard !items.isEmpty else {
```

```
            return nil
        }
        return items.removeLast()
    }

    func map<U>(_ f: (Element) -> U) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }
}
...
```

map(_:)方法只声明了一个占位类型U，但是用到了Element和U。能用Element是因为map(_:)在Stack内部，使得占位类型Element可用。map(_:)的方法体几乎和myMap(_:_:)一样，只是返回值是Stack而不是数组。试一下新方法，如代码清单22-9所示。

代码清单22-9　使用map(_:)

```
...
var intStack = Stack<Int>()
intStack.push(1)
intStack.push(2)
var doubledStack = intStack.map { 2 * $0 }

print(intStack.pop()) // 打印Optional(2)
print(intStack.pop()) // 打印Optional(1)
print(intStack.pop()) // 打印nil

print(doubledStack.pop()) // 打印Optional(4)
print(doubledStack.pop()) // 打印Optional(2)
...
```


22.3　类型约束

在写泛型函数和数据类型时有一件重要的事要记在心里，那就是在默认情况下我们对将要使用的具体类型一无所知。前面创建了整型和字符串的栈，但是也可以创建其他任何类型的栈。对具体类型所知甚少造成的一个实际影响就是我们对于具体类型的值能做的事情很少。举个例子，我们无法检查两个值是否相等。这段代码无法通过编译：

```
func checkIfEqual<T>(_ first: T, _ second: T) -> Bool {
    return first == second
}
```

这个函数可以以任何类型被调用，包括那些无法比较的类型，比如闭包。（很难描述两个闭包“相等”是什么意思。Swift不允许进行这种比较。）

如果不能对占位类型做任何假设的话，泛型函数就没什么用了。为了解决这个问题，Swift允许使用类型约束（type constraint）对传递给泛型函数的具体类型进行一些限制。有两种类型约

束：一种是类型必须是给定类的子类，还有一种是类型必须符合一个协议（或者一个协议组合）。

举个例子，Equatable是Swift提供的协议，用来声明两个值的相等性可以检查。（第25章会详细介绍Equatable。）要看类型约束如何起作用，可以写一个checkIfEqual(_:_:)函数，包含T必须符合Equatable的约束，如代码清单22-10所示。

代码清单22-10　使用类型约束以便检查相等性

```
...
func checkIfEqual<T: Equatable>(_ first: T, _ second: T) -> Bool {
    return first == second
}

print(checkIfEqual(1, 1))
print(checkIfEqual("a string", "a string"))
print(checkIfEqual("a string", "a different string"))
```

为了声明占位类型符合Equatable，我们用了和第19章一样的: Protocal语法。这样就能检查传入这个函数的实例的相等性。

每个占位类型都可以有一个类型约束。举个例子，写一个函数检查两个CustomString-Convertible值是否有相同的描述，如代码清单22-11所示。

代码清单22-11　使用类型约束检查CustomStringConvertible值

```
...
func checkIfDescriptionsMatch<T: CustomStringConvertible, U: CustomStringConvertible>(
        _ first: T, _ second: U) -> Bool {
    return first.description == second.description
}

print(checkIfDescriptionsMatch(Int(1), UInt(1)))
print(checkIfDescriptionsMatch(1, 1.0))
print(checkIfDescriptionsMatch(Float(1.0), Double(1.0)))
```

T和U都必须是CustomStringConvertible的约束保证了first和second都有返回字符串的属性description。（如果不符合，编译器会报错。）即使两个参数类型不同，还是可以比较它们的描述。

22.4　关联类型协议

知道了类型、函数和方法都可以是泛型的，你就自然会问：协议是不是也可以是泛型的？答案是“不可以”。不过，协议支持类型的一个相关特性：*关联类型*（associated types）。

我们来看一组Swift标准库定义的协议，探索一下关联类型的协议。我们即将用到的两个协议是IteratorProtocol和Sequence，它们共同实现自定义类型对利用for-in循环遍历的支持。首先来看看IteratorProtocol协议：

```
protocol IteratorProtocol {
    associatedtype Element
```

```
    mutating func next() -> Element?
}
```

IteratorProtocol协议只需要一个mutating方法next()，这个方法返回一个Element?值。有了IteratorProtocol，只要重复调用next()就可以不断产生新值。如果迭代器无法再产生新值了，next()就会返回nil。

在协议内部，associatedtype Element表示符合这个协议的类型必须提供具体类型作为Element类型。符合这个协议的类型应该在其定义内部为Element提供typealias定义。在playground的开头新建一个符合IteratorProtocol的结构体StackIterator，如代码清单22-12所示。

代码清单22-12 创建StackIterator

```
import Cocoa

struct StackIterator<T>: IteratorProtocol {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element? {
        return stack.pop()
    }
}

struct Stack<Element> {
    ...
}
...
```


StackIterator把Stack封装起来，并且弹出栈顶数据项来产生值。next()返回的Element的类型是T，所以要相应设置一下类型别名。

下面来看一下StackIterator的实际使用。新建一个栈，添加一些数据项，再创建一个迭代器，然后循环遍历它的值，如代码清单22-13所示。

代码清单22-13 使用StackIterator

```
...
var myStack = Stack<Int> ()
myStack.push(10)
myStack.push(20)
myStack.push(30)

var myStackIterator = StackIterator(stack: myStack)
while let value = myStackIterator.next() {
    print("got \(value)")
}
```

StackIterator有点冗余。因为Swift可以推断协议的关联类型，所以只要指明next()返回的是T?，就可以删除显式的类型别名，如代码清单22-14所示。

代码清单22-14 精简StackIterator

```
Import Cocoa

struct StackIterator<T>: IteratorProtocol {
    typealias Element = T

    var stack: Stack<T>

    mutating func next() -> Element? T? {
        return stack.pop()
    }
}
...
```

下一个要学习的关联类型协议是Sequence。Sequence的定义很长，但是关键部分很短：

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    func makeIterator() -> Iterator
}
```

Sequence的关联类型名为Iterator。: IteratorProtocol语法是关联类型的类型约束，其含义和泛型的类型约束一样：对于一个符合Sequence的类型来说，必须有一个符合IteratorProtocol协议的关联类型Iterator。Sequence还要求符合它的类型实现一个方法——makeIterator()。这个方法返回一个关联类型IteratorProtocol的值。因为我们已经有了适合栈的生成器，所以把Stack改为符合Sequence，如代码清单22-15所示。

代码清单22-15 让Stack符合Sequence

```
...
struct Stack<Element>: Sequence {
    var items = [Element]()

    mutating func push(_ newItem: Element) {
        items.append(newItem)
    }

    mutating func pop() -> Element? {
        guard !items.isEmpty else {
            return nil
        }
        return items.removeLast()
    }

    func map<U>(_ f: (Element) -> U) -> Stack<U> {
        var mappedItems = [U]()
        for item in items {
            mappedItems.append(f(item))
        }
        return Stack<U>(items: mappedItems)
    }
```

```
        func makeIterator() -> StackIterator<Element> {
            return StackIterator(stack: self)
        }
    }
    ...
```

这里又用到Swift的类型推断省去了显式的typealias Iterator = StackIterator<Element>，不过写出来也没错。

Sequence协议是Swift为实现for-in循环在内部使用的协议。既然Stack也符合Sequence协议，那就可以循环遍历其内容了，如代码清单22-16所示。

代码清单22-16　循环遍历myStack

```
...
var myStackIterator = StackIterator(stack: myStack)
while let value = myStackIterator.next() {
    print("got \(value)")
}

for value in myStack {
    print("for-in loop: got \(value)")
}
```


每次调用next()都会使StackIterator弹出栈中的值，这是一个具有相当大破坏性的操作。（注意for-in循环输出的元素顺序和元素的出栈顺序一样——都是把元素入栈的顺序倒过来。）当StackIterator从next()返回nil时，栈就是空的。不过，我们可以从myStack手动创建一个迭代器，然后在for-in循环中再次使用myStack。能这样重用是因为Stack是值类型，也就意味着每次StackIterator被创建时得到的都是栈的副本，原实例不会改变。

最后要注意一点：如果协议有关联类型，那么这个协议就不能用作具体类型。举个例子，不能声明一个IteratorProtocol类型的变量，也不能声明一个参数类型是IteratorProtocol的函数，因为IteratorProtocol有关联类型。不过，有关联类型的协议对于在泛型声明中使用where子句至关重要。

22.5　类型约束中的 where 子句

写一个方法，取出数组的每个元素并推入栈中，如代码清单22-17所示。

代码清单22-17　把数组中的数据项推入栈

```
...
struct Stack<Element>: Sequence {
    ...
    mutating func pushAll(_ array: [Element]) {
        for item in array {
            self.push(item)
        }
    }
```

```
}

...
for value in myStack {
    print("for-in loop: got \(value)")
}

myStack.pushAll([1, 2, 3])
for value in myStack {
    print("after pushing: got \(value)")
}
```

pushAll(_:)有用，但是还可以更通用。我们已经知道了任何符合Sequence的类型都可以在for-in循环中使用，那么这个方法为什么需要数组呢？它应该能接受任何序列类型——即使是另外一个Stack，因为Stack符合Sequence。

不过，我们的第一次尝试会产生编译错误，如代码清单22-18所示。

代码清单22-18　还差一点就对了

```
...
struct Stack<Element>: Sequence {
    ...
    mutating func pushAll(_ array: [Element]) {
    mutating func pushAll<S: Sequence>(_ sequence: S) {
        for item in array sequence {
            self.push(item)
        }
    }
}
...
```

我们用占位类型S把pushAll(_:)变成泛型方法，它是符合Sequence协议的类型。S的约束保证我们可以用for-in语法循环遍历之。不过，这还不够。为了把从sequence中取出的数据项推入栈，需要确保从序列类型中来的数据项类型和栈元素的类型匹配。也就是说，还需要一个约束让S所产生元素的类型是Element。

Swift用where子句支持这种约束，如代码清单22-19所示。

代码清单22-19　使用where子句来保证类型一致

```
...
struct Stack<Element>: Sequence {
    ...
    mutating func pushAll<S: Sequence>(_ sequence: S)
                where S.Iterator.Element == Element {
        for item in sequence {
            self.push(item)
        }
    }
}
...
```

pushAll(_:)是一个有占位类型的泛型方法。这个占位类型S有一个约束，那就是使用的具体类型必须符合Sequence协议。（记住这一点，因为这是Stack上的方法。Stack自己也是泛型类型，有一个占位类型Element，pushAll(_:)也能访问到占位类型Element。）

在占位类型后面的where子句执行进一步的约束。S.Iterator.Element引用了关联到Iterator类型的Element类型，而Iterator类型又是关联到S的。约束S.Iterator.Element == Element表示关联类型Element所用的具体类型必须和Stack的占位类型Element所用的一致。

泛型where子句的语法乍一看很难读懂，但是举个例子应该就能好懂很多。如果栈保存的是整型元素，那么传递给pushAll(_:)的参数必须是生成整型的序列类型。已知的两种生成整型的序列类型是Stack<Int>和[Int]。下面试试看，如代码清单22-20所示。

代码清单22-20　把数据项推入栈

```
...
var myOtherStack = Stack<Int>()
myOtherStack.pushAll([1, 2, 3])
myStack.pushAll(myOtherStack)
for value in myStack {
    print("after pushing items onto stack, got \(value)")
}
```


这段代码新建了一个空的整数栈myOtherStack，然后把一个数组中的整数全部推入myOtherStack，最后把myOtherStack中的所有整数推入myStack。两种情况之所以可以用同一个泛型方法，是因为数组和栈都符合Sequence。

泛型是Swift非常强大的功能，如果还没有完全理解也不要苦恼——这是一个既复杂又抽象的概念。慢慢来，回去看一遍本章写的Stack类，然后试试完成挑战练习。

22.6　青铜挑战练习

为Stack结构体添加filter(_:)方法。它应该接受一个参数，这个参数是接受一个Element并返回布尔型的闭包；然后返回一个新的Stack<Element>，包含闭包返回为真的元素。

22.7　白银挑战练习

写一个泛型函数findAll(_:_:)。这个函数接受一个符合Equatable协议的任意类型T的数组，以及一个元素（也是类型T）。findAll(_:_:)应该返回一个整数数组，对应这个元素在数组中出现的位置。举个例子，findAll([5,3,7,3,9], 3)应该返回[1, 3]，因为数据项3出现在数组索引1和3的位置。分别用整数和字符串测试你的函数。

22.8　黄金挑战练习

修改白银挑战练习中的findAll(_:_:)函数，让它接受一个泛型类型Collection而不是数

组。提示：你需要把返回值从[Int]改成Collection协议的关联类型的数组。

22.9　深入学习：理解可空类型

可空类型是所有非凡Swift程序的核心组成部分，而且这门语言有很多特性让使用可空类型比较容易。不过，Optional背后的原理却没什么特别的，它只有两个成员的泛型枚举而已：

```
enum Optional<Wrapped> {
    case None
    case Some(Wrapped)
}
```

你可能已经猜到了，None成员值对应当前值为nil的可空实例，而Some成员值对应值为类型Wrapped的可空实例。因为Some成员值是泛型，所以可以为任何类型创建一个可空版本。

大部分与可空类型的交互都会用到可空实例绑定和可空链式调用，但是也可以把可空类型当成其他枚举来交互。举个例子，如果maybeAnInt是Int?，那就可以针对两种成员值进行匹配：

```
switch maybeAnInt {
case .None:
    print("maybeAnInt is nil")

case let .Some(value):
    print("maybeAnInt has the value \(value)")
}
```

一般没有这个必要，但是知道可空类型其实没有魔法也挺好的。可空类型只不过是基于你也能用的Swift特性所构建的而已。

22.10　深入学习：参数多态

在第15章，我们学习了类继承。任何期望某个类为参数的函数也都可以接受这个类的子类作为参数。接受类或子类作为参数的这种能力一般被称为*多态*（polymorphism），不过更准确地说应该是*运行时多态*（runtime polymorphism）或者*子类多态*（subclass polymorphism）。多态的意思是“多种形式”，可以让一个函数接受不同的类型。

运行时多态是一个强大的工具，苹果为开发提供的框架经常会用到它。不幸的是，它也有缺点。有继承关系的类都紧密地联系在一起：很难改变一个而不影响另一个。由于编译器对函数接受参数的实现方式，运行时多态也有虽然小却可觉察的性能损失。

Swift为泛型添加约束的能力带来了另一种形式的多态：*编译时多态*（compile-time polymorphism），也被称为*参数多态*（parametric polymorphism）。带约束的泛型函数也符合多态的定义：一个函数可以接受不同的类型。

编译时多态函数能解决上述困扰运行时多态的两个问题。很多不同的类型可以符合一个协议，使它们能用在需要参数符合该协议的任何泛型函数上；但是这些类型可能毫不相关。这样可以很容易做到改变其中一个而不影响其他的。此外，编译时多态一般没有性能损失。

我们在playground中用数组和栈各调用一次`pushAll(_:)`。编译器实际上会在可执行文件中产生两个版本的`pushAll(_:)`，这意味着方法自己不需要在运行时做任何事情来处理不同的参数类型。

Swift正在进入一个有大量使用类继承和运行时多态传统的社区。不过，泛型和编译时多态正在开始发挥重要作用。下次开始写类继承时，先考虑一下要解决的问题是不是用协议和泛型解决更好。第23章会讨论更多为基于协议设计（protocol-based design）而服务的工具。

第 23 章

协议扩展

在过去几十年里，占统治地位的软件设计思想是面向对象编程（object-oriented programming，OOP）。OOP强大而且为人熟知，人们能够凭直觉认识到这种风格对代码意味着什么。传统上，OOP用类来为数据建模，用方法来修改这些类的实例的属性以及与其他类的实例进行通信。Swift支持OOP，但是支持的方式不是很传统，因为在OOP中枚举和结构体可以取代类的很多典型用法。

Swift还解决了一些OOP的缺陷。在OOP中，尤其要慎用继承，因为过深的继承层次很容易让代码充满难以理解的类。Swift为设计可重用、可组合的类型带来了新机会：不用类和继承，而是用协议和泛型。即便使用值类型，协议也能解决OOP中继承能解决的问题。协议扩展（protocol extension）是使得这种设计成为可能的强大工具。

23.1 为锻炼建模

开始探索协议扩展之前，需要一个协议和若干符合这个协议的类型做些实验。下面写一些基本的代码来记录锻炼计划。

新建playground，命名为ProtocolExtensions。从Exercise协议开始，如代码清单23-1所示。

代码清单23-1 Exercise协议

```
import Cocoa

var str = "Hello, playground"

protocol Exercise {
    var name: String { get }
    var caloriesBurned: Double { get }
    var minutes: Double { get }
}
```

Exercise协议有三个可读属性，分别用作锻炼的名字、消耗的热量（卡路里）和锻炼时长（分钟）。命名遵循了Swift标准库的惯例，在标准库中，协议名是名词（比如Exercise），或者对于表述能力的协议，则以-able、-ible或-ing这三种后缀结束。目前为止出现过的协议也都遵循这个命名惯例，比如Sequence、Equatable和CustomStringConvertible。

新建两个结构体记录锻炼：一种是椭圆机，另一种是跑步机（如代码清单23-2所示）。

代码清单23-2　EllipticalWorkout和TreadmillWorkout锻炼

```
...
protocol Exercise {
    var name: String { get }
    var caloriesBurned: Double { get }
    var minutes: Double { get }
}

struct EllipticalWorkout: Exercise {
    let name = "Elliptical Workout"
    let caloriesBurned: Double
    let minutes: Double
}

struct TreadmillWorkout: Exercise {
    let name = "Treadmill Workout"
    let caloriesBurned: Double
    let minutes: Double
    let laps: Double
}
```

这里新建的两个结构体都符合Exercise。每个结构体的名字都是常量，对所有实例都一样。它们的caloriesBurned和minutes属性都会在创建实例时被设置。TreadmillWorkout还有一个laps属性记录跑步的圈数。laps属性不需要符合Exercise，但是回忆一下第19章的内容：多余的属性和方法完全没问题。

给每个结构体新建一个实例，如代码清单23-3所示。

代码清单23-3　EllipticalWorkout和TreadmillWorkout的实例

```
...
struct EllipticalWorkout: Exercise {
    let name = "Elliptical Workout"
    let caloriesBurned: Double
    let minutes: Double
}

let ellipticalWorkout = EllipticalWorkout(caloriesBurned: 335, minutes: 30)

struct TreadmillWorkout: Exercise {
    let name = "Treadmill Workout"
    let caloriesBurned: Double
    let minutes: Double
    let laps: Double
}

let runningWorkout = TreadmillWorkout(caloriesBurned: 350, minutes: 25, laps: 10.5)
```

有了协议和符合协议的类型，可以开始为这些类型增加功能了。

23.2 扩展 Exercise

关于Exercise的实例，有一个很自然的问题就是：每锻炼一分钟消耗多少卡路里的热量？利用关于泛型和where子句的知识就可以写一个函数来计算这个值，如代码清单23-4所示。

代码清单23-4　用通用的方法计算每分钟消耗多少卡路里

```
...
func caloriesBurnedPerMinute<E: Exercise>(for exercise: E) -> Double {
    return exercise.caloriesBurned / exercise.minutes
}

print(caloriesBurnedPerMinute(for: ellipticalWorkout)))
print(caloriesBurnedPerMinute(for: runningWorkout))
```

caloriesBurnedPerMinute(for:)是一个泛型函数，它的占位类型必须是符合Exercise协议的类型。把caloriesBurnedPerMinute(for:)变成泛型函数就可以用任何符合Exercise协议的类型的实例作为参数调用它，包括EllipticalWorkout和TreadmillWorkout。函数体用了Exercise的两个属性计算每分钟消耗多少卡路里。

caloriesBurnedPerMinute(for:)本身没什么问题，但是如果拿到一个Exercise的实例，你必须记得有caloriesBurnedPerMinute(for:)这么一个函数存在。Exercise有calories-BurnedPerMinute属性会更自然，但是你肯定不想把同样的代码复制粘贴到EllipticalWorkout和TreadmillWorkout（以及以后有可能新建的Exercise类型）中去。

因此，取而代之的方法是在Exercise协议上写一个扩展来添加这个属性，如代码清单23-5所示。

代码清单23-5　为Exercise添加caloriesBurnedPerMinute

```
...
func caloriesBurnedPerMinute<E: Exercise>(for exercise: E) -> Double {
    return exercise.caloriesBurned / exercise.minutes
}
extension Exercise {
    var caloriesBurnedPerMinute: Double {
        return caloriesBurned / minutes
    }
}
...
```

与非协议扩展一样，协议扩展用的关键字也是extension。协议扩展可以添加有实现的计算属性和方法，但是不会增加协议的需求。正如协议不支持存储属性一样，协议扩展也不能添加存储属性。与写泛型函数时的限制相似，协议扩展内的实现只能访问其他肯定存在的属性和方法，比如本例中的caloriesBurned和minutes。所有符合某个协议的类型都能使用在该协议的扩展中添加的属性和方法。

删除caloriesBurnedPerMinute(for:)函数，现在playground会显示错误。用对calories-

BurnedPerMinute属性的访问替换这个函数的调用，如代码清单23-6所示。

代码清单23-6 访问caloriesBurnedPerMinute

```
...
print(caloriesBurnedPerMinute(for: ellipticalWorkout))
print(caloriesBurnedPerMinute(for: runningWorkout))
print(ellipticalWorkout.caloriesBurnedPerMinute)
print(runningWorkout.caloriesBurnedPerMinute)
```

结果是一样的。

23.3 带 where 子句的协议扩展

扩展可以给任何类型添加方法和计算属性，即使不是你写的类型也是如此。类似地，协议扩展能给任何协议添加方法和计算属性。不过，我们之前说过，协议扩展中添加的属性和方法只能使用肯定存在的其他属性和方法。

还记得第22章中的内建协议Sequence吗？它有一个associatedtype，名为Iterator。这个类型别名本身必须符合IteratorProtocol，而IteratorProtocol有一个名为Element的associatedtype表示生成器生成的元素类型。

在写Sequence的协议扩展时，没有多少有用的属性和方法。用where子句可以限制协议扩展只对Element是某个类型的Sequence生效。

23

为Sequence写一个协议扩展，限制它只对元素类型为Exercise的序列生效，如代码清单23-7所示。

代码清单23-7 扩展包含Exercise的Sequence

```
...
extension Sequence where Iterator.Element == Exercise {
    func totalCaloriesBurned() -> Double {
        var total: Double = 0
        for exercise in self {
            total += exercise.caloriesBurned
        }
        return total
    }
}
```

协议扩展的where子句语法和泛型一样。我们添加了一个totalCaloriesBurned()方法来计算序列中所有锻炼记录的总消耗卡路里数。在实现中，我们循环遍历self中的每个锻炼记录；这样可行是因为self是一种Sequence。然后访问每个元素的caloriesBurned属性；这样可行是因为where子句限制这个方法只对元素类型是Exercise的序列类型有效。

要使用这个扩展，创建一个Exercise的数组。数组符合Sequence，所以可以调用新的totalCaloriesBurned()方法，如代码清单23-8所示。

代码清单23-8　对一个ExerciseType数组调用totalCaloriesBurned()方法

```
...
extension Sequence where Iterator.Element == Exercise {
    func totalCaloriesBurned() -> Double {
        var total: Double = 0
        for exercise in self {
            total += exercise.caloriesBurned
        }
        return total
    }
}

let mondayWorkout: [Exercise] = [ellipticalWorkout, runningWorkout]
print(mondayWorkout.totalCaloriesBurned())
```

这个数组之所以能用totalCaloriesBurned()方法是因为它是[Exercise]类型，因此结果是685.0。反之，如果创建的数组是[Int]，就不能用totalCaloriesBurned()方法。Xcode的自动补全不会出现这个方法，但是如果手动输入，那么程序就不能编译。

23.4　用协议扩展提供默认实现

到目前为止，我们写的两个协议扩展都会给协议添加属性或方法。我们也可以利用协议扩展提供协议自身需求的默认实现。

回忆第19章的CustomStringConvertible协议，它有一个需求：一个可读的字符串属性description。改变Exercise，让它继承CustomStringConvertible，如代码清单23-9所示。这意味着它需要description属性。

代码清单23-9　让Exercise继承CustomStringConvertible

```
import Cocoa

protocol Exercise: CustomStringConvertible {
    var name: String { get }
    var caloriesBurned: Double { get }
    var minutes: Double { get }
}
...
```

playground现在有两个错误，因为EllipticalWorkout和TreadmillWorkout都没有description属性。

我们可以回过头去给这两个类型添加description，但是这样显得有点蠢，因为Exercise已经有了足够的属性，可以提供一个合理的字符串表示。用协议扩展为所有符合Exercise的类型添加description的一个默认实现，如代码清单23-10所示。

代码清单23-10　为Exercise添加description的一个默认实现

```
protocol Exercise: CustomStringConvertible {
```

```
    var name: String { get }
    var caloriesBurned: Double { get }
    var minutes: Double { get }
}

extension Exercise {
    var description: String {
        return "Exercise(\(name), burned \(caloriesBurned) calories
                in \(minutes) minutes)"
    }
}
...
```

playground中的错误消失了。扩展提供了`description`的一个默认实现，所以不需要符合`Exercise`的类型自己提供了。

打印两个`Exercise`实例来查看它们的描述，如代码清单23-11所示。

代码清单23-11　尝试默认的description实现

```
...
print(ellipticalWorkout.caloriesBurnedPerMinute)
print(runningWorkout.caloriesBurnedPerMinute)

print(ellipticalWorkout)
print(runningWorkout)
...
```


在playground的调试区域，你会看到如下输出。它跟`description`的实现完全吻合。

```
Exercise(Elliptical Workout, burned 335.0 calories in 30.0 minutes)
Exercise(Treadmill Workout, burned 350.0 calories in 25.0 minutes)
```

当协议为部分（或所有）属性或方法提供默认实现时，符合这个协议的类型就不需要再对其进行实现了。但是如果默认实现不合适，符合协议的类型也可以选择自己实现。

`TreadmillWorkout`还知道跑步的距离，但是这部分信息没有包含在描述中。在`TreadmillWorkout`上实现`description`属性，这个实现的优先级会高于`Exercise`的扩展提供的默认实现。从代码风格上说，把这个实现从`TreadmillWorkout`的核心功能中剥离出来放进扩展会更清晰。这种剥离在实际的应用中随处可见，即使把扩展定义在被扩展类型所在的同一个文件中也是一样，因为这样能很容易看清楚为了符合一个协议有哪些属性和方法被添加到了这个类型上。

代码清单23-12　覆盖协议的默认实现

```
...
struct TreadmillWorkout: Exercise {
    let name = "Treadmill Workout"
    let caloriesBurned: Double
    let minutes: Double
    let laps: Double
}

extension TreadmillWorkout {
```

```
        var description: String {
            return "TreadmillWorkout(\(caloriesBurned) calories and
                   \(laps) laps in \(minutes) minutes)"
        }
    }
    ...
```

既然TreadmillWorkout自己实现了description，那么从输出中应该能够看出，只有打印ellipticalWorkout的时候使用了默认实现。

```
Exercise(Elliptical Workout, burned 335.0 calories in 30.0 minutes)
TreadmillWorkout((350.0 calories and 4.2 miles in 25.0 minutes)
```

23.5　关于命名：一个警世故事

协议扩展有个边界情况，如果不小心的话，很容易让人深感挫败。在上一节中， 我们在Exercise的需求中增加了description（通过让Exercise继承CustomStringConvertible协议实现），为description添加了一个默认实现，并为TreadmillWorkout添加了一个优先级比默认实现高的特定实现。这样能正确运行是因为Exercise需要description。不过，如果写一个协议扩展来添加一个属性或方法，然后又给符合协议的类型添加一个名字相同（但实现不同）的属性或方法，会发生什么？

答案取决于访问实例的方式：编译器知道这个实例的特定类型，还是只知道它是协议的实例？这听起来有点让人困惑，不过没有关系。我们来看个例子。

用协议扩展实现Exercise上的title属性。打印mondayWorkout中所有锻炼记录的标题，如代码清单23-13所示。

代码清单23-13　扩展Exercise，添加标题

```
...
extension Exercise {
    var title: String {
        return "\(name) - \(minutes) minutes"
    }
}

for exercise in mondayWorkout {
    print(exercise.title)
}
```

我们添加了一个title实现，包含Exercise的名字和时长。代码应该会有如下输出：

```
Elliptical Workout - 30.0 minutes
Treadmill Workout - 25.0 minutes
```

现在回过头去在EllipticalWorkout上实现title属性。EllipticalWorkout实例的标题是锻炼所用椭圆机的品牌名，如代码清单23-14所示。

代码清单23-14　为EllipticalWorkout添加标题

```
...
struct EllipticalWorkout: Exercise {
    let name = "Elliptical Workout"
    let title = "Workout using the Go Fast Elliptical Trainer 3000"
    let caloriesBurned: Double
    let minutes: Double
}
...
```

看一下for循环的输出。

```
Elliptical Workout - 30.0 minutes
Treadmill Workout - 25.0 minutes
```

没有发生变化。为了确认没有输入错误，试着直接打印ellipticalWorkout的标题，如代码清单23-15所示。

代码清单23-15　打印ellipticalWorkout的标题

```
...
for exercise in mondayWorkout {
    print(exercise.title)
}

print(ellipticalWorkout.title)
```

应该会有如下输出：

```
Elliptical Workout - 30.0 minutes
Treadmill Workout - 25.0 minutes
Workout using the Go Fast Elliptical Trainer 3000
```


令人惊讶！同一个ellipcitalWorkout值在打印到控制台的时候给出了两种不同的title值。在代码清单23-13中，title打印到控制台是Elliptical Workout - 30.0 minutes。在代码清单23-15中，title打印出来是Workout using the Go Fast Elliptical Trainer 3000。似乎EllipticalWorkout提供的实现的优先级并没有Exercise的扩展的实现优先级高。为什么会这样？

有两个原因造成了这个结果。首先，title对于Exercise协议来说不是必需的。我们只是通过协议扩展为title提供了一个默认实现。这意味着当编译器看到代码清单23-13中的ellipticalWorkout.title时，会把ellipticalWorkout看作Exercise类型的实例。（回忆一下，我们对mondayWorkout进行了循环遍历，它的类型是[Exercise]。）

因此，编译器会使用它所知对于Exercise可用的title。编译器并没有去检查实际类型EllipticalWorkout的实现，因为title对于Exercise来说不是必需的。

其次，编译器会遵循我们提供的类型信息。在代码清单23-13中，我们告诉编译器数组中的实例是Exercise类型，所以它会用协议扩展提供的title的默认实现。在代码清单23-15中，我们告诉编译器实例是EllipticalWorkout类型的。这个类型提供了title的实现，所以编译器就

会用这个版本的title。

再重复一遍，如果感到困惑也是正常的。最重要的是要理解：在考虑写一个协议扩展添加属性或方法时，如果它们不是协议所需的属性或方法的默认实现，那就得小心了。如果符合协议的类型也实现了这些属性或方法，那么运行时行为可能就不是你所预期的。

23.6　青铜挑战练习

把title属性引入的杂乱代码清理干净。为Exercise协议添加title，确保输出和你预期的一样。

23.7　黄金挑战练习

这个挑战练习的独特之处在于没有特定的问题或答案。它旨在鼓励读者阅读苹果的Swift团队所写的接口。记住，在Xcode中按住Command键并点击类型、函数、方法甚至操作符可以跳转到展示元素声明方式的视图中。

第一次遇到map(_:)方法是在第13章，我们对数组调用了这个方法。不过map(_:)不只是数组的方法。map(_:)定义在Swift标准库中所有Sequence的一个协议扩展中。

Swift标准库包含大量由协议扩展提供的属性和方法，其中很多都包含where子句，以基于不同的准则限制其使用。

Swift标准库利用了很多我们学习过的高级特性。一开始可能很难读懂，尤其是在Swift是你第一次接触编程或第一次接触泛型的时候。不过，值得花一些时间看看标准库是如何组织的。试着在playground中按住Command键并点击Sequence，大致浏览一下那里定义的一些扩展。看看自己能不能明白某些where子句的含义。做些实验，探索一下！

第 24 章

内存管理和ARC

所有的计算机程序都会使用内存。大部分计算机程序会动态使用内存：程序在运行时动态分配和释放内存。Swift对内存管理的态度相对独特。它会自动处理好大部分内存问题，但是并没有使用垃圾回收器（程序语言中自动内存管理的常用工具）。与之相反，Swift使用的是引用计数系统。本章会研究这个系统如何工作，并学习避免内存泄漏所需的知识。

24.1 内存分配

值类型（枚举和结构体）的内存分配和管理很简单。新建值类型的实例时，系统会自动为实例划出大小合适的内存。任何传递实例的动作，包括传递给函数以及存储到属性中，都会创建实例的副本。当实例不再存在时，Swift会回收内存。你不需要做任何事情来管理值类型的内存。

本章要介绍的是引用类型（特别是类）的内存管理。新建类的实例时，系统会为实例分配内存，跟值类型一样。不过，区别在于传递类实例时发生的事情。把类实例传递给函数或存储到属性中会对同一块内存创建新的引用，而不是复制实例本身。对同一块内存有多个引用意味着，只要任何一个引用修改了类实例，所有的引用就都能看到变化。

Swift不像C那样需要手动管理内存，而是为每个类实例维护一个*引用计数*（reference count）。这是对组成类实例的内存的引用数量。只要引用计数大于0，实例就会存活。一旦引用计数变成0，实例就会被回收，`deinit`方法运行。

就在不久以前，用Objective-C开发的应用还在用手动引用计数。手动引用计数需要程序员管理所有类实例的引用计数。每个类都有一个方法来*保持*（retain）对象（增加引用计数）和一个方法来*释放*（release）对象（减少引用计数）。正如你想象中的那样，手动引用计数是很多bug的根源：如果保持一个实例太多次，那么这个实例可能就无法释放了（造成所谓的内存泄漏）；但是如果释放一个实例太多次，则会造成崩溃。

2011年，苹果为Objective-C引入了*自动引用计数*（automatic reference counting，ARC）。在ARC下，编译器负责分析代码并在所有合适的位置插入保持和释放调用。Swift也是在ARC的基础上构建的。你不需要做任何事情来管理类实例的引用计数——编译器会帮你做。不过，理解系统如何运作还是很重要的。有很多常见的错误会造成内存管理问题。

24.2　循环强引用

新建一个命令行工具，命名为CyclicalAssets。为工程添加新文件Person.swift，并插入如代码清单24-1所示的Person类定义。

代码清单24-1　定义Person类（Person.swift）

```
import Foundation

class Person: CustomStringConvertible {
    let name: String

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(self) is being deallocated")
    }
}
```

Person类只有一个属性，我们在初始化方法中为其设置了值。通过实现description计算属性，它符合CustomStringConvertible协议。我们还添加了deinit实现，以便当一个人被“释放”（即因为引用计数降到0而导致内存被回收）时可以看到。

现在，修改main.swift，创建一个可空Person，如代码清单24-2所示。

代码清单24-2　创建可空Person（main.swift）

```
import Foundation

print("Hello, world!")
var bob: Person? = Person(name: "Bob")
print("created \(bob)")

bob = nil
print("the bob variable is now \(bob)")
```

这里新建了一个Person?，打印出它的名字，再将其设置为nil。（让这个变量是可空的，这样就可以将其设置为nil，从而看到deinit的执行。）构建并运行程序，应该能看到如下输出：

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
Program ended with exit code: 0
```

变量bob是可空类型，持有一个类的实例（引用类型）。默认情况下，所有的引用都是强引用（strong reference），意味着它们会增加被指向实例的引用计数。因此，名字为Bob的Person在被

创建并赋给bob变量后的引用计数是1。把bob置为nil时，Bob的引用计数减少了。然后可以看到Person(Bob) is being deallocated的消息，因为引用计数降为了0。

接着，新建Swift文件Asset.swift，并插入Asset类，如代码清单24-3所示。

代码清单24-3 定义Asset类（Asset.swift）

```
import Foundation

class Asset: CustomStringConvertible {
    let name: String
    let value: Double
    var owner: Person?

    var description: String {
        if let actualOwner = owner {
            return "Asset(\(name), worth \(value), owned by \(actualOwner))"
        } else {
            return "Asset(\(name), worth \(value), not owned by anyone)"
        }
    }

    init(name: String, value: Double) {
        self.name = name
        self.value = value
    }

    deinit {
        print("\(self) is being deallocated")
    }
}
```


Asset类非常类似于Person类。Asset有name和value属性，符合CustomStringConvertible，并且会在被释放时打印消息。它还有一个变量存储属性owner，会指向拥有资产的Person。owner是可空的，因为可能存在无人认领的资产。

在main.swift中创建一些资产，如代码清单24-4所示。

代码清单24-4 创建资产（main.swift）

```
import Foundation

var bob: Person? = Person(name: "Bob")
print("created \(bob)")

var laptop: Asset? = Asset(name: "Shiny Laptop", value: 1_500.0))
var hat: Asset? = Asset(name: "Cowboy Hat", value: 175.0)
var backpack: Asset? = Asset(name: "Blue Backpack", value: 45.0)

bob = nil
print("the bob variable is now \(bob)")

laptop = nil
```

```
hat = nil
backpack = nil
```

这段代码又用到了可空类型，这样就可以把实例置为`nil`。这个操作会触发`deinit`方法。构建并运行应用。不出所料，所有的资产都被回收了，而且没有所有者：

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

人可以拥有东西；利用资产属性，`Person`类会模拟这种关系。回到Person.swift，添加一个属性和一个方法来让人获得资产，如代码清单24-5所示。

代码清单24-5　让Person拥有资产（Person.swift）

```
import Foundation

class Person: CustomStringConvertible {
    let name: String
    var assets = [Asset]()

    var description: String {
        return "Person(\(name))"
    }

    init(name: String) {
        self.name = name
    }

    deinit {
        print("\(self) is being deallocated")
    }

    func takeOwnership(of asset: Asset) {
        asset.owner = self
        assets.append(asset)
    }
}
```

这段代码添加了这个人所拥有的`Asset`的数组`assets`，还有把资产交给这个人的方法`takeOwnership(of:)`。获取资产所有权意味着这个人把资产加入自己的`assets`数组，并把资产的`owner`属性指向自己。在main.swift中，给Bob一组资产的所有权，如代码清单24-6所示。

代码清单24-6　Bob获取所有权（main.swift）

```
...
var laptop: Asset? = Asset(name: "Shiny Laptop", value: 1_500.0)
var hat: Asset? = Asset(name: "Cowboy Hat", value: 175.0)
var backpack: Asset? = Asset(name: "Blue Backpack", value: 45.0)
```

```
bob?.takeOwnership(of: laptop!)
bob?.takeOwnership(of: hat!)

bob = nil
...
```

构建并运行，输出可能有点出人意料：

```
created Optional(Person(Bob))
the bob variable is now nil
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

唯一被回收的实例是背包——当我们设置`backpack = nil`时，它的引用计数降到了0。笔记本电脑、帽子和Bob本人则没有被释放。为什么？看一下图24-1，这幅图显示了在main.swift中把人和变量值设置为`nil`之前都是谁引用了谁。

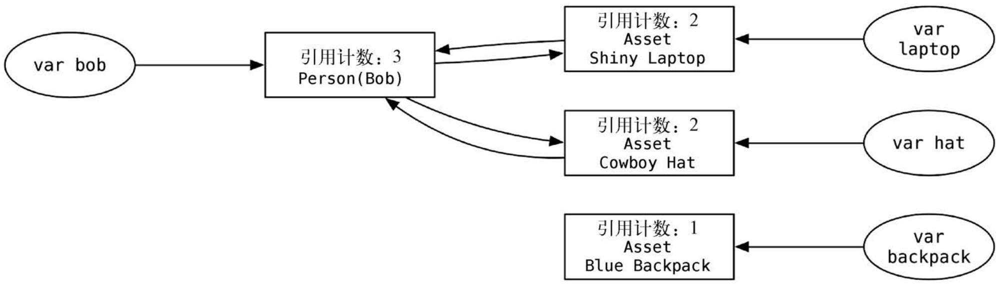


图24-1 CyclicalAssets（之前）

24

`Person`和`Asset`的每个实例都用方框表示，并标记有当前的引用计数。记住，`bob`本身不是实例，它只是引用了一个`Person`类的实例。方框中的引用计数是指向实例的箭头数量，也就是引用这个实例的数量。在main.swift中把所有的变量设置为`nil`后，这些引用就消失了，只留下图24-2中的这些。

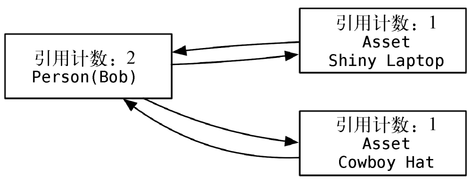


图24-2 CyclicalAssets（之后）

我们创建了两个循环强引用（strong reference cycle），这个术语表示两个实例互相强引用对方。Bob引用了笔记本电脑（通过`assets`属性），而笔记本电脑引用了Bob（通过`owner`属性）。Bob和帽子之间也一样。这些实例的内存已经无法访问了（指向这些实例的变量都没了），但是内

存不会被回收，因为每个实例的引用计数都大于0。

循环强引用就是一种*内存泄漏*（memory leak）。应用分配了足够Bob和其资产所需的内存，但是当程序不再需要这些内存后并没有将其还给系统。

不用担心这样会对电脑造成影响：CyclicalAssets这样的程序停止运行后，所有的内存（包括泄漏的内存）都会被操作系统回收。不过，内存泄漏还是个严重的问题；跟macOS比起来，对iOS更是如此。iOS记录每个应用使用的内存，应用如果使用太多的内存就会被杀掉。一个应用有内存泄漏时，被泄漏的内存仍然算在这个应用所用的总内存数中，即使它不再需要这些内存或者根本没有使用这些内存。

24.3　用 weak 打破循环强引用

循环强引用的一个解决办法就是打破循环。当把`bob`置为`nil`后，马上通过循环遍历资产并把所有者置为`nil`就可以手动打破循环，但是这种方法既麻烦又容易出错。Swift提供了一个关键字来达到同样的效果。修改`Asset`，把`owner`属性从强引用改为*弱引用*（weak reference），如代码清单24-7所示。

代码清单24-7　把owener属性改成弱引用（Asset.swift）

```
import Foundation

class Asset: CustomStringConvertible {
    let name: String
    let value: Double
    weak var owner: Person?
    ...
}
```

弱引用不增加所指向实例的引用计数。在本例中，把`owner`改为弱引用意味着当我们把Bob设置成笔记本电脑和帽子的所有者时，Bob的引用计数不会增加。Bob唯一的强引用就是main.swift中的`bob`变量。

现在，当我们把`bob`变量置为`nil`时，Bob的引用计数降为0，所以会被释放。当Bob被释放时，就不再持有对资产的强引用，所以资产的引用计数也会降为0。再次运行程序，确认所有的对象都被释放了。

```
created Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

如果一个弱引用指向的实例被释放会发生什么？弱引用会被置为`nil`。在main.swift中，当`bob`被置为`nil`前后输出一些日志信息就可以看到实际效果，如代码清单24-8所示。

代码清单24-8 谁拥有帽子（main.swift）

```
...
bob?.takeOwnership(of: laptop!)
bob?.takeOwnership(of: hat!)

print("While Bob is alive, hat's owner is \(hat!.owner)")
bob = nil
print("the bob variable is now \(bob)")
print("After Bob is deallocated, hat's owner is \(hat!.owner)")
...
```

再次运行程序，实际演示弱引用变量：

```
created Optional(Person(Bob))
While Bob is alive, hat's owner is Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
After Bob is deallocated, hat's owner is nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

弱引用有两个条件：

- 弱引用必须用`var`声明，不能用`let`；
- 弱引用必须声明为可空类型。

之所以存在这两个条件，都是因为一旦弱引用指向的实例被释放，弱引用就会变成`nil`。能变成`nil`的类型只有可空类型，所以弱引用必须是可空的。因为用`let`声明的实例不能变，所以弱引用必须用`var`声明。

在大多数情况下，可以很容易地避免刚才解决的这种循环强引用问题。`Person`是拥有资产的类，所以它持有对资产的强引用是合理的。`Asset`是`Person`拥有的类，如果它需要所有者的引用，就应该用弱引用。毕竟，是人拥有资产，而不是资产拥有人。

还有一种更复杂的情况会产生循环强引用：在闭包中捕获`self`。

24.4 闭包中的循环引用

是时候添加一个会计类来记录`Person`的净资产了。新建Swift文件Accountant.swift并定义新类，如代码清单24-9所示。

代码清单24-9 定义`Accountant`（Accountant.swift）

```
import Foundation

class Accountant {
    typealias NetWorthChanged = (Double) -> Void

    var netWorthChangedHandler: NetWorthChanged? = nil
```

```
        var netWorth: Double = 0.0 {
            didSet {
                netWorthChangedHandler?(netWorth)
            }
        }

        func gained(_ asset: Asset) {
            netWorth += asset.value
        }
    }
```

Accountant定义了一个类型别名NetWorthChanged，它是接受一个双精度浮点数（新的净资产值）且没有返回值的闭包。这个类有两个属性：netWorthChangedHandler和netWorth。前者是当净资产值变化时调用的一个可空闭包，而后者是一个人的当前净资产。netWorth有一个didSet属性观察者，如果netWorthChangedHandler非空的话就会调用它。最后，gained(_:)函数用来告诉会计某个资产的价值应该加到净资产值上。

更新Person.swift，给它一个记录Person净资产的会计，如代码清单24-10所示。

代码清单24-10　为Person添加Accountant（Person.swift）

```
    import Foundation

    class Person: CustomStringConvertible {
        let name: String
        let accountant = Accountant()
        var assets = [Asset]()

        var description: String {
            return "Person(\(name))"
        }

        init(name: String) {
            self.name = name

            accountant.netWorthChangedHandler = {
                netWorth in

                self.netWorthDidChange(to: netWorth)
                return
            }
        }

        deinit {
            print("\(self) is being deallocated")
        }

        func takeOwnership(of asset: Asset) {
            asset.owner = self
            assets.append(asset)
            accountant.gained(asset)
        }
```

```
        func netWorthDidChange(to netWorth: Double) {
            print("The net worth of \(self) is now \(netWorth)")
        }
    }
```

我们添加了一个accountant属性，默认值是一个新的Accountant实例。Person有对Accountant的强引用，这是完全合理的。在init()中，我们把会计的netWorthChangedHandler设置为调用新的netWorthDidChange(to:)方法，这个方法会打印此人新的净资产值。最后，更新takeOwnership(of:)，通知会计有新资产。构建并运行程序，你应该能看到如下输出：

```
created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
the bob variable is now nil
After Bob is deallocated, hat's owner is Optional(Person(Bob))
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

有一条净资产值变化的消息，刚才添加的会计代码似乎都正确运行了。不过，循环强引用问题又出现了：Bob、笔记本电脑和帽子都没有被释放。为什么这些实例没有从内存中删除呢？

新代码有一个不那么明显的循环强引用。Person有一个通过属性对Accountant的强引用，但是Accountant并没有对Person的强引用——至少第一眼看上去是这样的。提示一下，尝试修改Person的init()方法（这样会导致编译错误），如代码清单24-11所示。

代码清单24-11 修改init()（Person.swift）

```
...
    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            netWorth in

            self.netWorthDidChange(to: netWorth)
            return
        }
    }
```

现在试着构建程序。错误信息指出Call to method 'netWorthDidChange' in closure requires explicit 'self.' to make capture semantics explicit。闭包的捕获语义（capture semantics）是什么意思？

回忆一下第13章的内容，闭包能捕获在闭合作用域中定义的变量。默认情况下，闭包的捕获是通过对用到的变量的强引用实现的。netWorthDidChange(to:)是Person的一个方法，而它又是捕获了self的闭包。所以，在闭包内调用这个方法会让闭包对捕获的Person实例形成强引用——也就是self。这解释了为什么会有内存泄漏：Accountant实际上有对Person的强引用！Accountant的netWorthChangedHandler通过自己的Person的self强引用了这个Person，如图24-3所示。因为两个实例相互对对方强引用，就会存在一个循环，导致两者都不能被销毁。

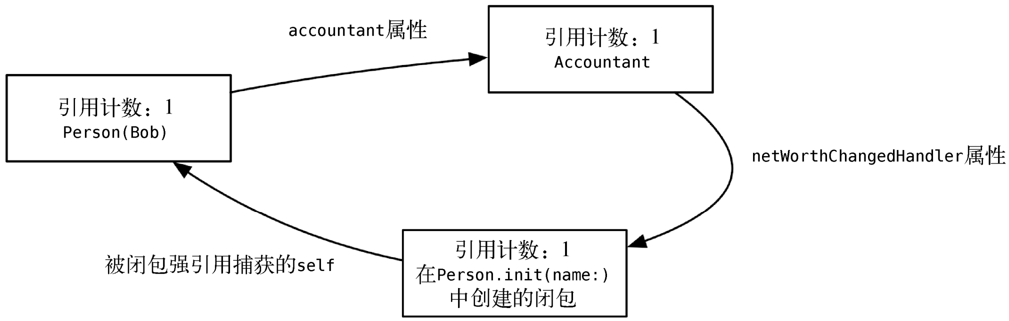


图24-3　Person对Accountant对Person的循环强引用

再看一眼错误信息：`... to make capture semantics explicit`。Swift本来可以隐式地在闭包中使用`self`，但是这么做很容易不小心造成循环强引用，如上面这个例子一样。因此，Swift通常需要我们显式地使用`self`，强迫我们考虑发生循环引用的可能性。（对于某些编译器知道不可能发生循环引用的情况，就没必要显式地使用`self`了。很快就会有例子说明。）

要改变闭包的捕获语义，使捕获变成弱引用，需要用捕获列表（capture list）。修改Person.swift，在创建闭包时使用捕获列表，如代码清单24-12所示。

代码清单24-12　使用捕获列表（Person.swift）

```
...
    init(name: String) {
        self.name = name

        accountant.netWorthChangedHandler = {
            [weak self] netWorth in

            self?.netWorthDidChange(to: netWorth)
            return
        }
    }
...
```

捕获列表的语法是在闭包参数列表前面加上带方括号的变量列表。这里的捕获列表告诉Swift弱引用捕获`self`，而不是强引用。现在`Accountant`的闭包不再强引用`Person`，循环强引用被打破了。

注意`self?`在闭包体中的使用。因为`self`是被弱引用捕获的，而所有的弱引用实例都必须是可空的，所以闭包内的`self`是可空的。

再次运行程序，确认所有的实例都被正确释放了。

```
created Optional(Person(Bob))
The net worth of Person(Bob) is now 1500.0
The net worth of Person(Bob) is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
```

```
After Bob is deallocated, hat's owner is nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

24.5　逃逸闭包和非逃逸闭包

Swift还支持不可能产生循环强引用的闭包。这类闭包被称为非逃逸闭包（non-escaping closure），不需要显式引用`self`。以函数参数形式声明的闭包默认是非逃逸的；其他场景中的闭包是逃逸的，比如`netWorthChangedHandler`这样的属性。

逃逸（escaping）是什么意思？逃逸表示传递给一个函数的闭包可能会在该函数返回后被调用。也就是说，闭包逃脱出了接收它作为参数的函数的作用域。如果闭包是非逃逸的，编译器就能知道它不可能在函数返回后被调用，所以不可能产生循环强引用。

我们来看一个实际中的例子。假设我们不想让一个`Person`完全得到资产，直到其`Accountant`达到资产的价值。给`Accountant`的`gained(_:)`方法传递一个`completion`闭包就可以解决这个问题，如代码清单24-13所示。

代码清单24-13　给`gained(_:)`添加一个`completion`闭包（Accountant.swift）

```
import Foundation

class Accountant {
    ...
    func gained(_ asset: Asset, completion: () -> Void) {
        netWorth += asset.value
        completion()
    }
}
```


当会计增加客户的资产净值后，就调用`completion`闭包。更新`Person`的`takeOwnership(of:)`方法，把对个人资产的改动移动到`completion`闭包里，如代码清单24-14所示。

代码清单24-14　使用`completion`闭包（Person.swift）

```
import Foundation

class Person: CustomStringConvertible {
    ...
    func takeOwnership(of asset: Asset) {
        asset.owner = self
        assets.append(asset)
        accountant.gained(asset) {
            asset.owner = self
            assets.append(asset)
        }
    }
    ...
}
```

注意，不需要写`self.assets.append(asset)`。编译器知道传递给`gained(_:completion:)`的闭包是非逃逸的，所以可以隐式地引用`self`的属性和方法。构建并运行应用，输出应该和上次运行是一样的。

如何告诉编译器需要让闭包是逃逸的？对`Person`添加一个方法，使调用者为这个人的净资产变化设置一个不同的处理方法闭包，如代码清单24-15所示。（首次尝试将不会编译，不过还是试一下来看看会出现什么错误。）

代码清单24-15　添加`useNetWorthChangedHandler(_:)`（Person.swift）

```
import Foundation

class Person: CustomStringConvertible {
    ...
    func useNetWorthChangedHandler(handler: (Double) -> Void) {
        accountant.netWorthChangedHandler = handler
    }
}
```

这样会产生一个错误，提示你在混合使用逃逸闭包和非逃逸闭包（如图24-4所示）。

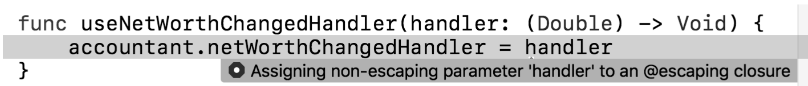


图24-4　传递非逃逸参数的错误

我们试图把`handler`赋给`accountant.netWorthChangedHandler`属性。把闭包存到属性中意味着可以在函数返回后调用它，也就是说闭包会逃脱出函数的作用域。因为闭包默认是非逃逸的，所以编译器拒绝执行这种赋值操作。

把`handler`标记为逃逸闭包可以修复这个错误，如代码清单24-16所示。

代码清单24-16　把`handler`标记为逃逸闭包（Person.swift）

```
import Foundation

class Person: CustomStringConvertible {
    ...
    func useNetWorthChangedHandler(handler: @escaping (Double) -> Void) {
        accountant.netWorthChangedHandler = handler
    }
}
```

`@escaping`属性告诉编译器`handler`可能逃脱出`useNetWorthChangedHandler(_:)`方法。这样就修复了上面这个错误。

回到main.swift使用这个新方法，如代码清单24-17所示。

代码清单24-17　使用`useNetWorthChangedHandler(_:)`（main.swift）

```
...
bob?.useNetWorthChangedHandler { netWorth in
```

```
    print("Bob's net worth is now \(netWorth)")
}
bob?.takeOwnership(of: laptop!)
bob?.takeOwnership(of: hat!)
...
```

构建并运行应用就能看到新的资产净值处理方法实际工作了。

```
created Optional(Person(Bob))
Bob's net worth is now 1500.0
Bob's net worth is now 1675.0
While Bob is alive, hat's owner is Optional(Person(Bob))
Person(Bob) is being deallocated
the bob variable is now nil
After Bob is deallocated, hat's owner is nil
Asset(Shiny Laptop, worth 1500.0, not owned by anyone) is being deallocated
Asset(Cowboy Hat, worth 175.0, not owned by anyone) is being deallocated
Asset(Blue Backpack, worth 45.0, not owned by anyone) is being deallocated
Program ended with exit code: 0
```

24.6　青铜挑战练习

Person的资产所有权概念还不完整。Person能通过某种方式获取资产所有权，但是没有办法放弃资产所有权。更新Person，使得一个实例可以放弃一项资产。（提示：如果想让净资产值不出错，可能也要更新Accountant。）

24.7　白银挑战练习

在main.swift中再创建一个Person。在把笔记本电脑的所有权给Bob后，试着马上把同一个笔记本电脑的所有权给这个新的Person。现在两个人都拥有了这台笔记本电脑！修复这个bug。

24.8　深入学习：我能获取实例的引用计数吗

不幸的是，Swift没有开放对任何实例的实际引用计数的访问。（虽然第18章提到过我们可以通过isKnownUniquelyReferenced(_:)函数知道一个变量是否是对某个实例的唯一引用。）

即使可以知道实例的引用计数是多少，答案也可能跟你预期的不一样。在本章中，我们说过类似于“在这里，引用计数是2”的话。这是个善意的谎言。

从概念上说，把引用计数理解为我们描述的那样完全没问题。在底层，编译器可以随意添加保持（增加引用计数）调用和释放（减少引用计数）调用。只要结果正确，就对程序没有坏处。如果可以询问一个实例的实际引用计数，那么答案取决于编译器在这个地方做了什么分析。此外，在苹果的系统库中有些类在引用计数上的行为比较奇怪（具体细节超出了本书的范围）。

对你而言，真正重要的是如何识别潜在的循环强引用以及如何用weak打破它。

第 25 章 Equatable和Comparable

很多时候，编程需要依赖于值的比较。知道两个值是否相等是很重要的；如果不相等，还要知道一个值比另一个值大还是小。

事实上，在本书中，我们已经对Swift的基本类型这么做了。这个字符串和那个字符串相等吗？这个整数比那个整数小吗？所有的Swift基本类型实例都知道自己如何跟同类型的其他实例进行比较。为什么？

答案跟值类型存在的目的密切相关。这些类型的实例表示某些值。人们有一个固有的期望，那就是值能够而且应该是可比较的。我们本能地想知道一个整数和另一整数相比谁更大。

基于这个原因，让自定义值类型知道如何与其他实例进行比较是一个很好的做法。Swift为测试相等性和可比性提供两个协议：Equatable和Comparable。本章会展示如何让自定义类型符合这些协议，其中包括实现一些函数。这些函数告诉我们的实例如何跟同类型的其他实例进行比较。新建名为Comparison的playground，开始本章的学习。

25.1 符合 Equatable

新建一个不符合Equatable的类型，如代码清单25-1所示。

代码清单25-1　定义Point

```
import Cocoa

var str = "Hello, playground"

struct Point {
    let x: Int
    let y: Int
}
```

上面的结构体定义了Point类型。Point的x和y属性描述二维坐标系中的位置。

当前，Point还不知道如何判断一个实例是否等于另一个实例。新建两个该类型的实例，观察一下在查看两者是否相等时会发生什么事，如代码清单25-2所示。

代码清单25-2　创建两个Point实例

```
...
struct Point {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
```

我们用编译器自带的成员初始化方法新建了两个点a和b。我们给这两个点设置了一样的x和y属性。现在试着用==运算符测试两个点是否相等，如代码清单25-3所示。

代码清单25-3　a和b一样吗

```
...
struct Point {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
```

对相等性进行检查的这段代码无法运行。事实上，这还会产生一个编译错误。这个错误的根源在于我们还没有告诉Point结构体如何测试两个实例的相等性。

告诉结构体测试方法涉及让Point符合Equatable协议。把如代码清单25-4所示的代码添加到结构体声明中。

代码清单25-4　添加符合协议的声明


```
...
struct Point: Equatable {
    let x: Int
    let y: Int
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
```

现在又出现了一个新错误。这次，错误发生在声明Point结构体的那一行。错误消息是Point没有符合Equatable协议。简单地说，就是编译器不知道如何检查a和b是否相等。

为了弄清楚如何符合Equatable协议，打开文档。按住Option键并点击Equatable，在弹出的视图中点击底部的链接Protocol Reference，打开完整的参考文档。

文档告诉我们需要实现==运算符。但是==已经有定义了，事实上还是好几个定义。我们可以用==比较字符串、双精度浮点数、整数和字典等。因为Point是自定义类型，我们需要再提供一个==的实现，这样Swift就能知道如何比较这个类型的两个实例了。

Equatable协议指出符合这个协议的类型应该有一个静态方法==。实现==来比较Point类型两个实例，看它们的x和y属性的值是否相等来判断它们的相等性。如代码清单25-5所示。

代码清单25-5　修改==的实现

```
...
struct Point: Equatable {
    let x: Int
    let y: Int

    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
```

现在有了新的==运算符实现。（注意，运算符只是有特殊名字的函数。）这个定义有两个参数：lhs表示等号左边的实例，rhs表示等号右边的实例。两个参数的类型都是Point。

函数的实现很简单，就是比较通过函数参数传进来的两个Point实例的x值和y值，然后返回一个Bool表示两个实例是否相等。

看一下playground的运行结果侧边栏，你会看到错误消失了，我们成功测试了a和b的相等性。因为两个点的x值和y值相等，所以这两个点也是相等的。（试着改变一个点的x值，你就会看到两个点不相等了。继续往下读之前确保把改动恢复成原来的值。）

25.1.1　插曲：中缀运算符

把==声明为静态方法可能看起来有点奇怪。在第15章，我们讲到了静态方法是定义在类型上的，但是对==方法的调用并不在Point类型上，类似于Point.==(a, b)。事实上，==这样的运算符在Swift中是定义在全局层面的。

==在Swift标准库中的定义如下：

```
precedencegroup ComparisonPrecedence {
    higherThan: LogicalConjunctionPrecedence
}

infix operator == : ComparisonPrecedence
```

先不用管precedencegroup，25.7节会讨论这个话题。一定要注意==运算符被声明为中缀运算符。也就是说，在调用这个方法的时候，==运算符是放在要比较的两个实例中间的。因此，调用Point的==实现是像上例那样写的：(a == b)。

Swift编译器知道要在类型内寻找运算符的定义。举个例子，Equatable协议能够声明符合它的类型要实现一个static方法。如果创建自己的协议并把一个运算符作为对符合这个协议的类型的要求，而协议又没有把这个方法声明为static的，那么编译器就会产生一个警告。

```
protocol MyEquatable {
    func ==(lhs: Self, rhs: Self) -> Bool
    // Operator '==' declared in protocol must be 'static'
}
```

25.1.2　方法“买一赠一”

Point结构体现在符合Equatable了，所以就可以像刚才一样测试Point的相等性了。此外，Swift标准库基于==的定义提供了一个!=实现。一旦一个类型通过实现自己的==符合了Equatable，也就有了一个能工作的!=函数。

添加如代码清单25-6所示的测试来尝试这个函数。

代码清单25-6　a和b不相等吗

```
...
struct Point: Equatable {
    let x: Int
    let y: Int

    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
let abNotEqual = (a != b)
```

运行结果侧边栏应该会显示测试不相等性的结果是false。也就是说，这两个点是相等的。

25.2　符合 Comparable

既然Point符合了Equatable协议，你可能会对更精细的比较结果感兴趣。比如，一个点是否小于另一个点。符合Comparable协议可以实现这个功能。

打开Comparable的文档。因为还没有输入Comparable（无法按住Option键并点击它的名字），所以要点击Help菜单，选择Documentation and API Reference。搜索Comparable来看看需要做什么。你会发现需要实现一个运算符：中缀运算符（<）。在结构体中添加如代码清单25-7所示的代码，让它符合Comparable。

代码清单25-7　符合Comparale

```
...
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int
```

```
    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }

    static func <(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x < rhs.x) && (lhs.y < rhs.y)
    }
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
let abNotEqual = (a != b)
```

这段代码给Point添加了符合Comparable协议的声明，还添加了<运算符的实现。这个实现的工作方式和==差不多。它会检查左边传入的点是不是小于右边的点。如果左边点的x值和y值都小于右边的点，函数会返回true。否则，函数会返回false，表示左边的点不比右边的点小。

新建两个点来测试这个函数，如代码清单25-8所示。

代码清单25-8　测试<函数

```
...
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int

    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }

    static func <(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x < rhs.x) && (lhs.y < rhs.y)
    }
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == d)
let cLessThanD = (c < d)
```

这段代码用不同的x值和y值新建了两个点，然后查看c和d是否相等，结果是false：表示这两个点不相等。最后，练习<的用法，判断c是否小于d。在本例中，比较的结果是true。点c比点d小是因为它的x值和y值都比d的小。

跟符合Equatable协议一样，实现一个函数可以得到很多功能。Swift标准库用<和==运算符定义了>、>=和<=。这就是Comparable只需要我们实现<运算符的原因。如果类型符合

Comparable，就会自动得到这些运算符的实现。

添加一系列新的比较来测试这个功能，如代码清单25-9所示。

代码清单25-9　练习使用比较

```
...
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int

    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }

    static func <(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x < rhs.x) && (lhs.y < rhs.y)
    }
}

let a = Point(x: 3, y: 4)
let b = Point(x: 3, y: 4)
let abEqual = (a == b)
let abNotEqual = (a != b)
let c = Point(x: 2, y: 6)
let d = Point(x: 3, y: 7)

let cdEqual = (c == b)
let cLessThanD = (c < d)

let cLessThanEqualD = (c <= d)
let cGreaterThanD = (c > d)
let cGreaterThanEqualD = (c >= d)
```

最后三行比较检查以下结论是否成立：

- c小于等于d
- c大于d
- c大于等于d

正如我们预测的，这些比较结果分别是true、false和false。

25.3　继承 Comparable

Comparable实际上继承自Equatable。你可能已经猜到这种继承意味着什么了。为了符合Comparable协议，必须同时提供==的实现以符合Equatable。这种关系也意味着，如果一个类型声明符合Comparable，那就不需要显式声明符合Equatable。从Point结构体上删除显式声明符合Equatable的代码，如代码清单25-10所示。

代码清单25-10　删除不必要的符合声明

```
...
struct Point: Equatable, Comparable {
    let x: Int
    let y: Int

    static func ==(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x == rhs.x) && (lhs.y == rhs.y)
    }

    static func <(lhs: Point, rhs: Point) -> Bool {
        return (lhs.x < rhs.x) && (lhs.y < rhs.y)
    }
}
...
```

你应该能看到playground的运行结果跟之前一样。

关于风格，最后再说几句。虽然显式声明符合`Equatable`和`Comparable`并没有错，但是没有必要。如果一种类型符合了`Comparable`，那它一定也符合`Equatable`。这一点在文档中有明确说明，所以这是一个符合`Comparable`的类型的可预期行为。添加符合`Equatable`的声明没有增加太多信息。

另一方面，如果某种类型符合一个继承自其他协议的自定义协议，那么显式地继承所有涉及的协议是合理的。虽然这么做没有必要，但是可以让代码更具可读性和可维护性，因为你的自定义协议并没有在官方文档中说明。

25.4　青铜挑战练习

实现两个点相加。两个点的和应该返回一个新的`Point`，它会把两个点的`x`值和`y`值相加。需要实现一个接受两个`Point`实例的`+`运算符。

25.5　黄金挑战练习

新建一个`Person`类，它有两个属性：`name`和`age`。为方便起见，创建一个初始化方法，为两个属性提供参数。

接着，新建两个`Person`的实例，将其赋给两个常量`p1`和`p2`。再创建一个保存这些实例的数组`people`，然后把它们放进数组。

我们偶尔需要找出一个自定义类型的实例在一个数组中的索引。对数组调用`index(of:)`方法，参数是你想找出其索引的容器中某个元素的值。用这个方法找到`p1`在`people`数组中的索引。

你会发现一个错误。花些时间理解这个错误然后解决它。你应该能把`index(of:)`的结果赋给常量`p1Index`，它的值应该是`0`。

25.6 白金挑战练习

我们当前让Point符合Comparable的做法会产生一些令人困惑的结果。

```
let c = Point(x: 3, y: 4)
let d = Point(x: 2, y: 5)

let cGreaterThanD = (c > d) // 假
let cLessThanD = (c < d)    // 假
let cEqualToD = (c == d)    // 假
```

如上例所示，当一个点的x值和y值不是都大于另一个点时，麻烦来了。实际上，用这种方式比较两个点是不合理的。

改变Point符合Comparable的方式以修复这个问题。计算每个点离原点的欧几里得距离，而不是比较x值和y值。当a离原点的距离比b离原点的距离近时，这个实现应该对a < b返回true。

用如图25-1中的公式计算一个点的欧几里得距离。

$$distance(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$$

图25-1 欧几里德距离

25.7 深入学习：自定义运算符

Swift允许开发者创建自定义运算符。这个特性意味着我们可以创建自己的运算符来表示两个Person的实例结婚了。举个例子，我们想用+++函数来让两个人结婚。

创建Person类，如代码清单25-11所示。

代码清单25-11 创建Person类

```
...
class Person {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}
```

这个类有两个属性：一个表示名字，另一个表示配偶。它有一个给这些属性赋值的初始化方法。注意spouse属性是可空的，表示一个人可能没有配偶。

接着，创建这个类的两个实例，如代码清单25-12所示。

代码清单25-12 创建两个人的实例

```
...
class Person {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)
```

现在，声明一个新的中缀运算符；必须在全局作用域中声明。接着定义新的运算符函数如何工作，如代码清单25-13所示。

代码清单25-13 声明自定义运算符

```
...
class Person {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}
```

新的运算符+++用于让两个Person的实例结婚。作为一个中缀运算符，它用在两个实例中间。+++的实现会把每个实例赋给对方的spouse属性。

+++没有指定precedencegroup，也就是说使用DefaultPrecedence组。

```
precedencegroup DefaultPrecedence {
    higherThan: TernaryPrecedence
}
```

那么，上面代码中的higherThan是指什么？higherThan定义了运算符相对另一个precedencegroup的优先级。在本例中，DefaultPrecedence的优先级比另一个叫作TernaryPrecedence的precedencegroup高，这是三目运算符的precedencegroup。（第3章有

关于三目运算符的讨论）。所以我们的新运算符+++(lhs:rhs:)比三目运算符优先级高，会更早得到执行。

想一下 4 + 3 × 3 – 1 这个式子如何体现乘法比加法的优先级更高。因为乘法的优先级更高，所以 4 + 3 × 3 – 1 的结果是12。如果加法的优先级比乘法高，那么结果会是20。

precedencegroups提供了一些定义自定义运算符的其他选项。除了higherThan，还有一个重要的选项是associativity。

associativity定义了同一优先级里的运算符如何结合。它接受left或right。比如，运算符+和-属于同一个precedencegroup（AdditionPrecedence），所以优先级相同，两者都是左结合的。数学运算符的结合性和优先级决定了上面这个式子的执行顺序是 (4 + (3 × 3)) – 1。也就是说，先计算 3 × 3，因为乘法优先级最高，得到的结果是左结合的，所以得到了 (4 + 9) – 1，结果是 13 – 1，也就是12。

因为+++的作用是让两个Person实例结婚，所以不需要串接多次调用。比如，你不会看到这样的代码：matt +++ drew +++ someOtherInstance。因此，只要用默认的优先级和结合律就可以了。

练习使用你的新运算符，如代码清单25-14所示。

代码清单25-14 使用自定义运算符

```
...
class Person {
    var name: String
    weak var spouse: Person?

    init(name: String, spouse: Person?) {
        self.name = name
        self.spouse = spouse
    }
}

let matt = Person(name: "Matt", spouse: nil)
let drew = Person(name: "Drew", spouse: nil)

infix operator +++

func +++(lhs: Person, rhs: Person) {
    lhs.spouse = rhs
    rhs.spouse = lhs
}

matt +++ drew
matt.spouse?.name
drew.spouse?.name
```

代码matt +++ drew用于让这两个实例结婚。通过运行结果侧边栏查看这个过程是否可行。

虽然这个运算符能用，而且也不难看出它的作用，但我们一般还是建议避免声明自定义运算符。为自定义类型创建自定义运算符时，最好让每个读代码的人都能看懂。这意味着把自定义运

算符局限在众所周知的数学运算符范围内。（事实上，Swift只允许使用有明确定义的数学符号集创建自定义运算符。举个例子，你无法把+++重构成表示飞吻的emoji，也就是U+1F61A。）

未来某个查看代码的人可能会不明白+++是什么意思。（你自己也可能忘记。）毕竟它在这个例子中的含义相当模糊。

此外，这个自定义运算符也没有比marry(_:)方法更优雅、更高效地完成任务。举个例子，一个marry(_:)方法可能是这样的：

```
func marry(_ spouse: Person) {
    self.spouse = spouse
    spouse.spouse = self
}
```

这段代码的可读性更好，我们能很容易地看懂代码在做什么。这样会让代码在将来更容易维护。

Part 6

第六部分

事件驱动的应用

在这部分中，我们将应用前面所学的Swift知识开发第一个Mac应用和第一个iOS应用。随着学习过程的深入，你会了解有关Swift和Objective-C互操作的基本知识。

第 26 章

第一个Cocoa应用

Swift有一个很吸引人的特性是具备和Objective-C交互的能力，后者是写Mac和iOS应用的传统语言。我们不会完整讲解如何在一个应用中同时使用这两种语言，但是以下三章将会给你一个大致的体验。Swift使得调用Objective-C库成为可能，比如开发桌面Mac应用的原生API——Cocoa。

Swift能够利用被广泛描述为桥接（bridging）的技术实现和Cocoa（以及其他Objective-C框架）通信的能力。桥接就是从一种语言调用另一种语言中函数或实例的过程。桥接可以有两个方向：Swift能调用Objective-C函数，Objective-C也能调用Swift函数（有一些限制）。多数情况下，编译器会自动处理桥接的细节，不过偶尔需要我们介入，提供一些帮助。我们很快就能看到一些例子。

在本章中，我们要创建一个Mac的桌面应用VocalTextEdit。VocalTextEdit是一个非常简单的文本编辑器，还有一个朗读文档的特性。这个应用很简单，只是让你体验一下Cocoa开发。要彻底探索Cocoa，请参考最新版的《苹果开发之Cocoa编程》一书。

VocalTextEdit是基于文档的应用（document-based application），允许用户同时打开多个窗口，分别代表不同的文件。完成后，应用看起来如图26-1所示。

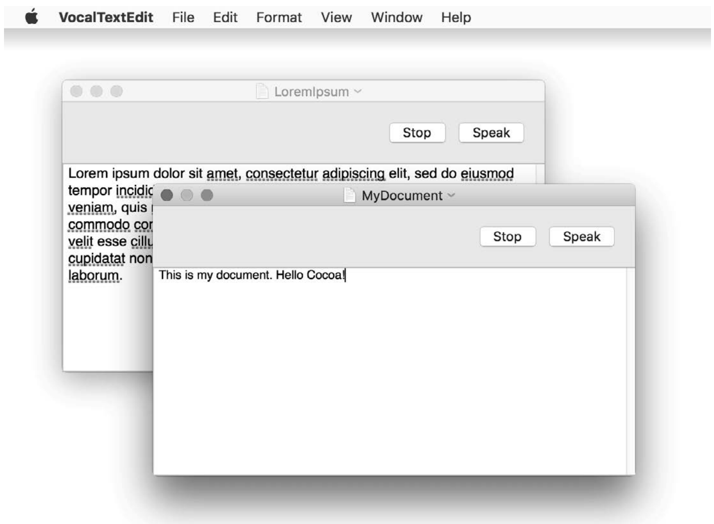


图26-1　完成后的VocalTextEdit应用

每个VocalTextEdit文档窗口的下半部分看起来都应该比较熟悉：就是一个简单的文本编辑器。顶部的Speak按钮会让电脑把文本文档的内容朗读出来。Stop按钮用于停止朗读。正如用户所预期的，作为基于文档的应用，VocalTextEdit也支持正常的保存、打开和自动保存功能。

26.1 开始创建 VocalTextEdit

打开Xcode，选择File → New → Project...（也可以在欢迎窗口点击Create a new Xcode project）。选中macOS区域（不是iOS区域），选择Application区域的Cocoa，点击Next（如图26-2所示）。

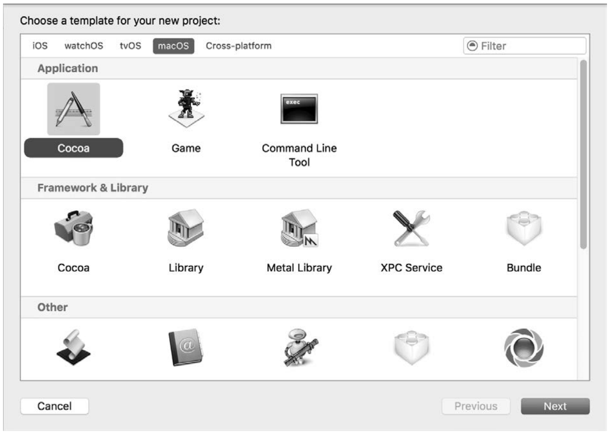


图26-2 选择Cocoa应用模版

在下一个窗口中，把工程命名为VocalTextEdit（如图26-3所示）。确保选择的语言是Swift。把复选框Use Storyboards和Create Document-Based Application都选中。输入txt作为Document Extension（VocalTextEdit最终就是一个文本编辑器，它会保存文本文件）。

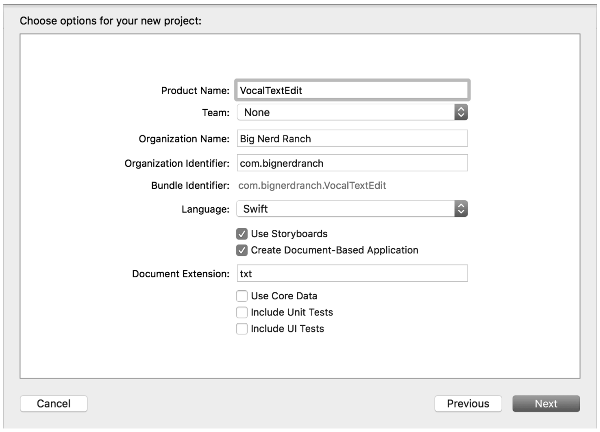


图26-3 配置VocalTextEdit

点击Next，将其保存到你想要的地方，完成工程的创建。

在继续之前，先来简单看一下Cocoa和iOS应用都会用到的核心设计模式。

26.2　模型–视图–控制器

模型–视图–控制器（model-view-controller，MVC）是一种设计模式。它的理念是，任何类的工作性质都不外乎以下三种：模型、视图和控制器。下面是对这三种分工的解释。

- 模型负责保存数据，并使其他对象能访问数据。模型不知道用户界面是怎样的，也不知道如何把自己绘制到屏幕上。它的主要目的是持有和管理数据。举个例子，一个记录学校出勤的应用会为`Student`定义一个模型对象。一个`Student`会为一个学生的所有属性建模，比如名字、年级等。一般用字符串和数组这样的Swift类型作为模型对象的组成部分。在VocalTextEdit中，`Document`会充当用户每个文本文件的模型对象。
- 视图是应用的可见元素。视图知道如何把自己绘制到屏幕上，也知道如何响应用户输入。视图不知道自己显示的实际数据是什么，也不知道数据如何组织和存储。一个简单的经验规则是：肉眼可见的就是视图。在VocalTextEdit中，视图对象包括`NSTextView`和`NSButton`的实例。
- 控制器负责控制视图和模型之间必要的逻辑。控制器会处理事件（通常来自于应用的用户），并把视图的信息传递给模型，反之亦然。控制器是任何应用的真正劳动力，因为它们是模型和视图的协调者。在VocalTextEdit中，`ViewController`会协调`Document`和屏幕上视图之间的关系。

图26-4展示了响应用户事件（比如点击按钮）的对象之间的控制流。注意，模型和视图不会直接通信——控制器在中间运筹帷幄，从某些对象接收消息，并给其他对象分发指令。

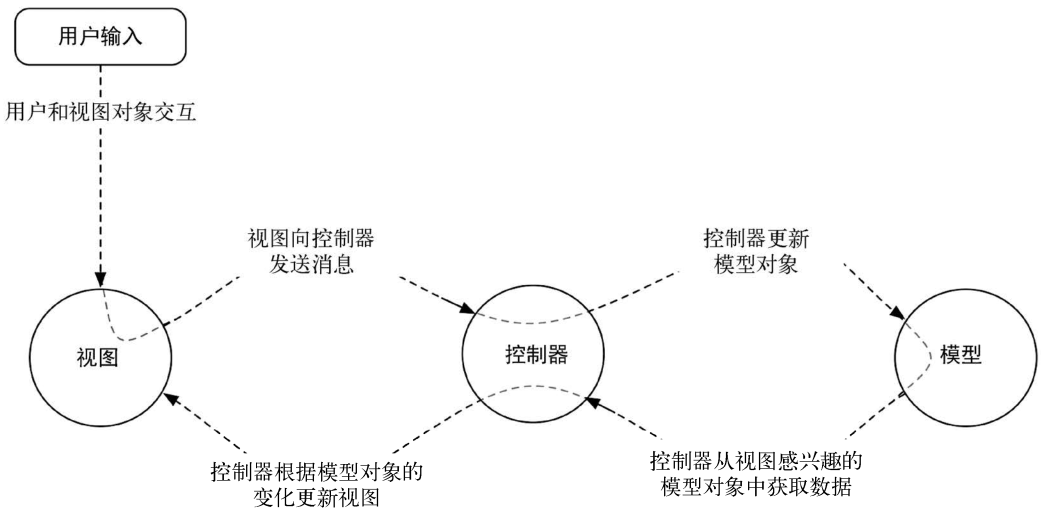


图26-4　用户输入的MVC控制流

26.3 设置视图控制器

模版创建了几个Swift文件，以及一个我们马上要用到的Main.storyboard。首先，打开ViewController.swift。如代码清单26-1所示，先把模版提供的几个覆盖函数（overriden function）删掉，因为这个应用用不到这些函数。

代码清单26-1 清理模版代码（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    override var representedObject: Any? {
        didSet {
        // Update the view, if already loaded.
        }
    }

}
```

注意，`ViewController`是`NSViewController`的子类。顾名思义，视图控制器负责管理用户界面并响应用户行为。

添加一个属性和两个实例方法，如代码清单26-2所示。有些代码看起来不太熟悉，我们会在后面解释。

代码清单26-2 添加一个属性和两个实例方法（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {

    @IBOutlet var textView: NSTextView!

    @IBAction func speakButtonClicked(_ sender: NSButton) {
        print("The speak button was clicked")
    }

    @IBAction func stopButtonClicked(_ sender: NSButton) {
        print("The stop button was clicked")
    }
}
```

先忽略`@IBOutlet`和`@IBAction`，看一下其他代码。我们声明了一个隐式展开的可空类型属性`textView`，两个接受`NSButton`且没有返回值的函数。`textView`属性指向文档窗口中显示可编

26

辑文本的部分。在macOS下，Cocoa提供这个功能的类是NSTextView。

按住Option键并点击NSTextView查看文档。在弹出的窗口中，点击底部的Class Reference打开NSTextView完整的参考文档（如图26-5所示）。

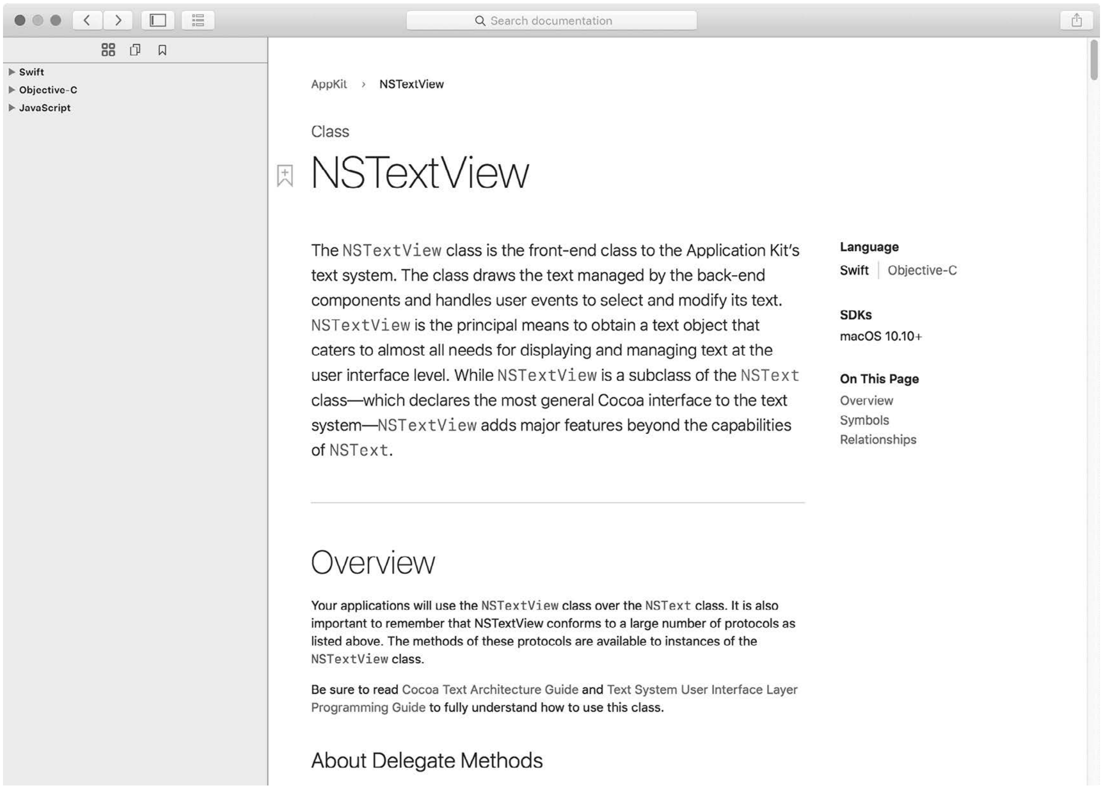


图26-5　NSTextView的参考文档

NSTextView有很多属性和方法，但是这个应用中唯一用到的是由NSTextView的父类NSText提供的。在NSTextView参考文档的Relationships一节，点击NSText链接可以跳转到NSText的参考文档。NSText提供了string属性，能让我们以字符串的形式获取和设置文本视图的内容。

@IBOutlet和@IBAction中的IB表示Interface Builder。很久以前（以软件的标准来看），苹果公司为开发应用提供了两个工具：用来写源代码的Xcode和用来布局用户界面的Interface Builder。Interface Builder后来被合并进了Xcode，但是程序员以及苹果还是把Xcode中跟“用户界面布局”有关的部分叫作Interface Builder。

@IBOutlet和@IBAction是标记。这里的@IBOutlet和@IBAction不会以任何有意义的方式改变代码，而是让Xcode知道被标记的属性和方法跟Interface Builder有关。@IBOutlet告诉Xcode，属性textView会在Interface Builder中被设置；而@IBAction告诉Interface Builder，这个方法会在用户跟视图交互时（比如点击按钮）被调用。

26.4　在 Interface Builder 中设置视图

现在来设置视图。在Xcode左侧的工程导航区域选择Main.storyboard。故事板文件以可视化的方式用Interface Builder布局应用的窗口和视图。大型应用可以使用多个故事板，而VocalTextEdit只会用Main.storyboard。Main.storyboard一开始看起来如图26-6所示。

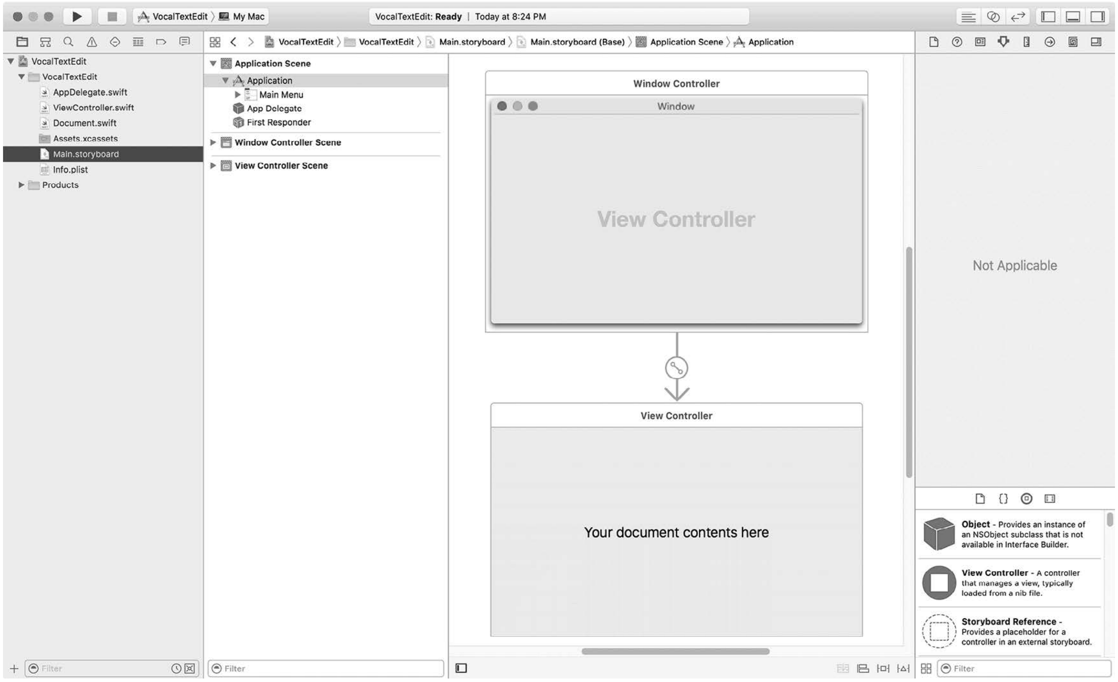


图26-6　Main.storyboard的初始内容

图26-6中用户界面左边的面板是文档大纲（document outline）。文档大纲以树的形式显示每个对象和子对象的关系。如果看不到文档大纲，点击故事板底部的Show Document Outline按钮（如图26-7所示）。

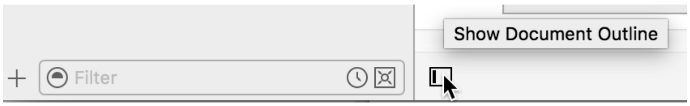


图26-7　显示文档大纲

现在Main.storyboard中有三块区域，从上到下分别是应用主菜单、窗口控制器布景和视图控制器布景。我们只会用到视图控制器布景（view controller scene）。

当前，视图控制器布景里有一行包含在`NSTextField`中的文本。我们不需要它，所以点击Your document contents here文本，按Delete键删除它。

现在，我们为应用准备好了空白的画布，接下来就要对VocalTextEdit进行布局了。如果工具区没有显示，点击Xcode右上角的按钮来显示它，如图26-8所示。

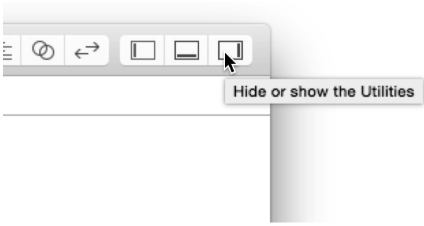


图26-8　显示Xcode的工具区

工具区的下半部分是库（library）。库分成了几个标签页，由不同的图标区分。选择◎图标显示对象库（object library）。对象库里的对象可以直接拖放到布局网格上构建用户界面。

26.4.1　添加朗读和停止按钮

在对象库底部的搜索框中搜索button。出现在对象库顶部的Push Button表示`NSButton`的一个实例。

要把按钮放在视图控制器的用户界面上，只要从对象库中拖动按钮到视图控制器布景上即可。当拖动按钮往视图的右上角移动时，会出现蓝色的虚线，按钮会吸附到图26-9中所示的位置。这些虚线被称为引导线，源自苹果的“人机界面指南”（*Human Interface Guidelines*，HIG）。HIG代表苹果用户界面设计制定的标准。苹果所有的平台都有HIG，可以在开发者文档中找到。

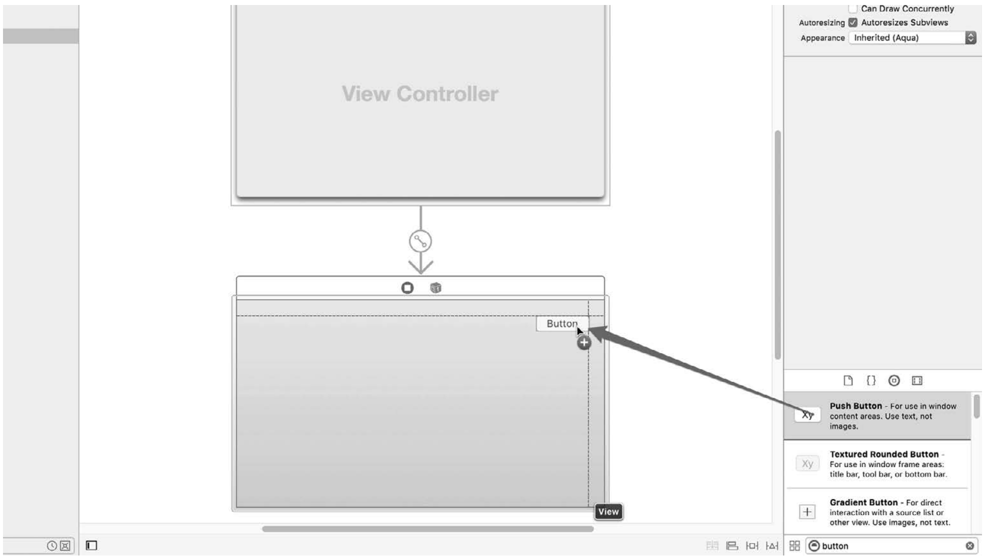


图26-9　从对象库中拖放对象到故事板

现在视图控制器布景中有按钮了，需要给它设置一个标题。双击按钮的文本，输入Speak。因为Speak比Button稍微短一些，所以按钮可能会稍微偏离位置。把它拖回角落，直至吸附到蓝色虚线上。（本章稍后会讨论如何解决这个问题。）

VocalTextEdit需要Speak和Stop按钮。再拖动一个按钮到视图控制器上。把第二个按钮重命名为Stop，拖动它直至吸附到Speak按钮左边的位置。现在视图控制器的布局看起来应该类似于图26-10。

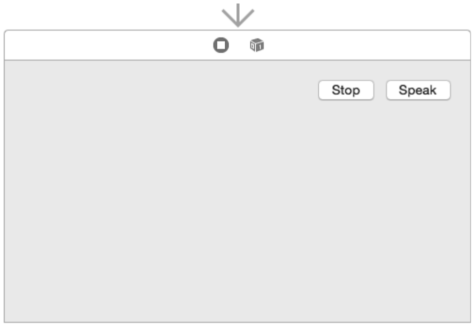


图26-10　VocalTextEdit的布局：两个按钮

26.4.2　添加文本视图

VocalTextEdit主要是一个文本编辑器，所以需要有一个区域让用户输入文本。回到对象库的搜索框，输入textview。把文本视图对象拖动到视图控制器上，放在按钮下面的空白区域中间，如图26-11所示。

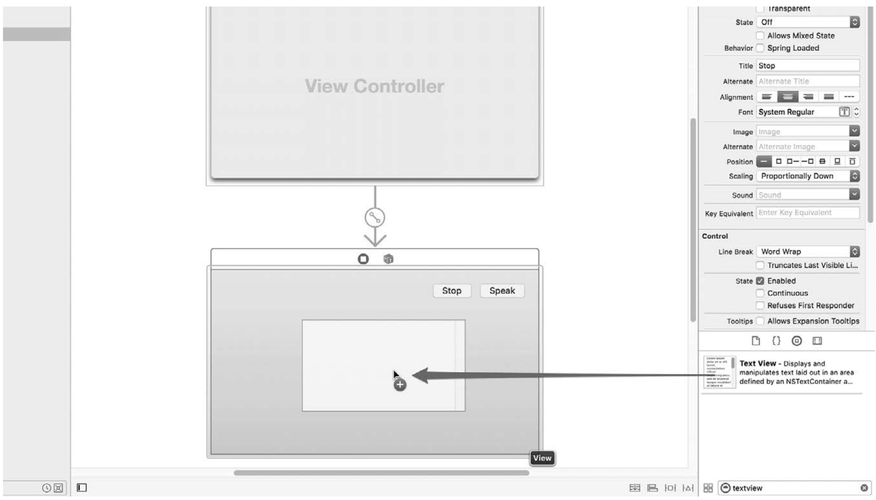


图26-11　VocalTextEdit布局：添加文本视图

我们可以在布局区域中点击元素来选中它们，也可以在文档大纲中点击来选中。在文档大纲中选择Bordered Scroll View - Text View，如图26-12所示。

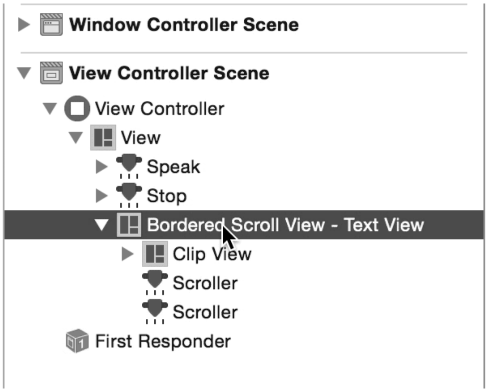


图26-12　在文档大纲中选中文本视图

注意，现在文本视图在文档大纲和视图控制器布景中都处于选中状态（前者是高亮，后者是在四条边和四个角上出现可拖动的方块）。利用文本视图边上的方块调整其大小。拖动左边、右边和下边直到和视图控制器的边缘对齐，拖动上边直到它吸附在之前添加的两个按钮下方。现在，布局看起来应该类似于图26-13。



图26-13　VocalTextEdit：布局

构建并运行应用。VocalTextEdit启动以后，可以用Command-N组合键（或选择File → New）新建文档，还可以在文本视图中输入文字。不出所料，Speak和Stop按钮没有什么作用。此外，还有错误弹出，称自动保存失败。我们会在本章结尾解决这些问题。

还有一个不太明显的问题。试着调整文档窗口的大小，以便有更多的空间来查看文档。你会发现界面不大对（如图26-14所示）。

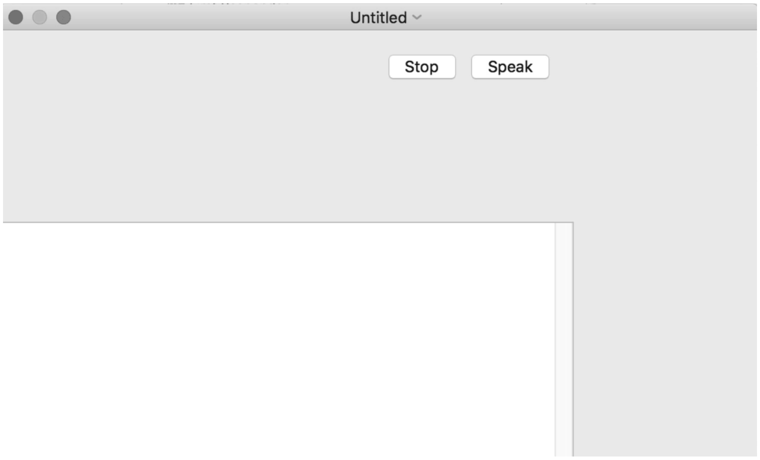


图26-14 调整大小失败

调整大小时这种不正常的行为肯定不是我们想要的。现在来修复这个问题。

26.4.3 自动布局

苹果提供了一个被称为自动布局的系统，可以设置一些约束来定义在布局计算过程中视图之间的关系。对自动布局的完整介绍超出了本书的范围，不过我们可以设置一些基本的约束，以便在调整VocalTextEdit的大小时保证其行为是合理的。

首先，创建约束强制Speak按钮处于文档窗口的右上角。按住Control键，点击Speak按钮，然后向右上方拖动，直到视图控制器的视图高亮，如图26-15所示。



图26-15 自动布局：Speak按钮

放开鼠标后，会弹出一个上下文菜单。我们要约束按钮的右边和上边，所以按住Shift键并点击Trailing Space to Container和Top Space to Container（如图26-16所示）。[在从左至右阅读的语言

（比如英语）中，“结束边缘”（trailing edge）是指视图的右边界。在从右至左阅读的语言（比如希伯来语）中，“结束边缘”则是指视图的左边界。“起始边缘”（leading edge）与之相反。]

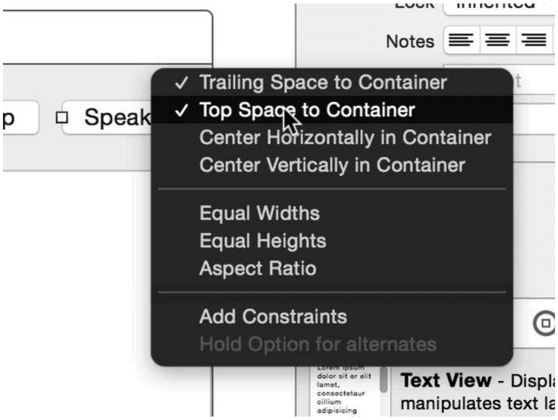


图26-16　自动布局：Speak按钮的约束

按下回车键，会看到一条蓝色的工字形线连接了Speak按钮的顶部和视图控制器视图的顶部，另一条则连接了右边。再次运行VocalTextEdit，尝试调整窗口大小。现在我们调整窗口大小的时候Speak按钮就会留在正确的位置（右上角）了。

不过Stop按钮和文本视图还是不对，需要给它们也加上约束。为Stop按钮添加跟Speak按钮一样的约束，把它钉在视图控制器视图的右上角。但是这样没有表达真正的布局关系：我们并不关心Stop按钮相对视图控制器视图的位置，而是想让它一直待在Speak按钮的左边。

要在Stop按钮和Speak按钮之间创建约束，需要按住Control键并拖动Stop按钮到Speak按钮上。出现弹出菜单后，按住Shift键并选择Horizontal Spacing和Baseline（如图26-17所示），然后按下回车键。

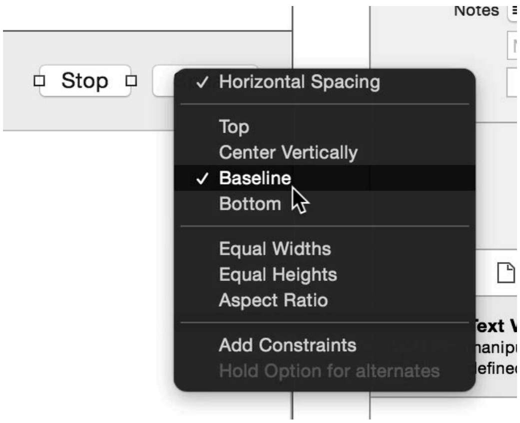


图26-17　自动布局：Stop按钮的约束

水平间距约束确保两个按钮之间的水平距离是固定的。基线约束确保两个按钮是按照按钮内的文本基线垂直对齐的。再次运行VocalTextEdit，调整窗口大小，确认Stop按钮一直处在正确的位置。

轮到文本视图了。我们要让文本视图的大小和窗口减去顶部两个按钮的空间后一样。还记得我们之前是怎么在文档大纲中选择元素的吗？在文档大纲中也可以创建约束。按住Control键，从Bordered Scroll View - Text View拖动到它的父视图，如图26-18所示。

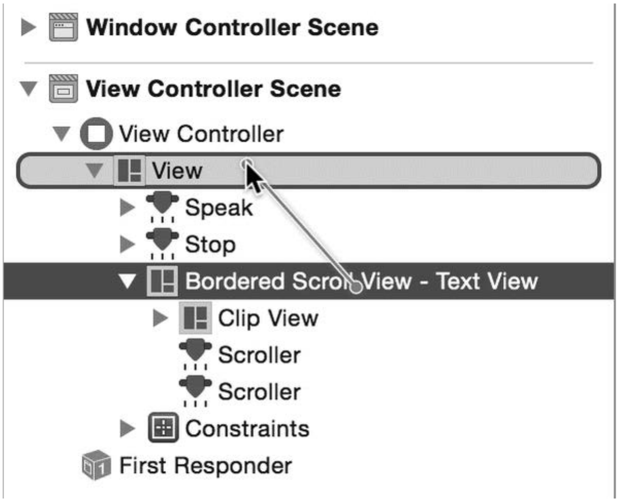


图26-18 自动布局：从文本视图到视图控制器的视图

在弹出的菜单中，按住Shift键并选择三个约束：Leading Space to Container、Trailing Space to Container和Bottom Space to Container。这些约束会钉住左边、右边和下边，使其紧贴窗口的边缘。

我们还需要约束上边。回到布局区域，按住Control键，拖动文本视图到Speak按钮，在弹出菜单中选择Vertical Spacing。这样会保持按钮和文本视图之间的距离不变。

再次运行VocalTextEdit并调整窗口大小，现在所有的界面元素都会合理响应窗口大小的变化。

26.5 连接

Interface Builder并不是只能创建视图、布局用户界面，它还能连接视图和Swift代码。

26.5.1 为 VocalTextEdit 的按钮设置目标−动作对

在运行应用并且加载Main.storyboard之后，Cocoa运行时会创建一个`ViewController`的实例，配置好视图，设置好出口（outlet）和动作。对于VocalTextEdit来说，我们希望对管理当前文档的`ViewController`实例调用`speakButtonClicked(_:)`方法。按住Control键，从Speak按钮拖动到表示`ViewController`的图标上，如图26-19所示。

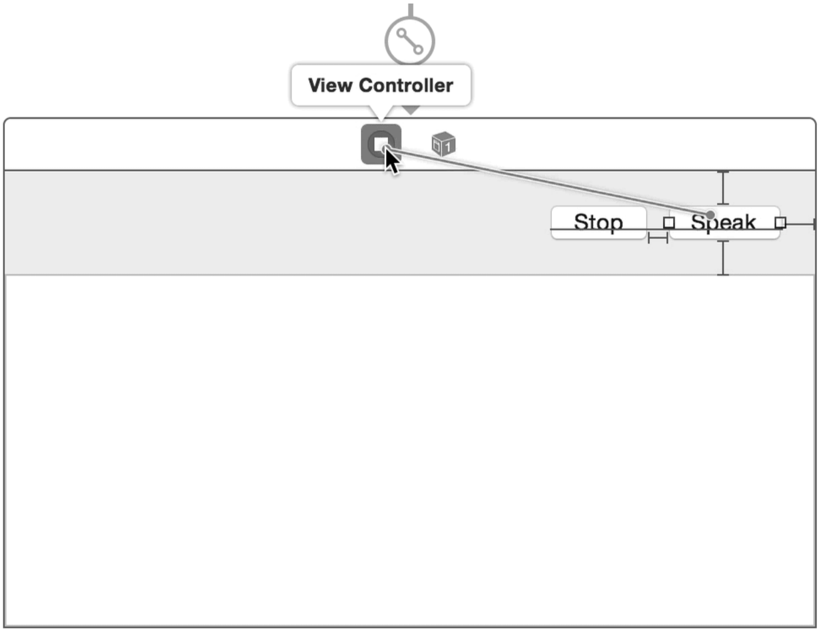


图26-19　连接Speak按钮

放开鼠标后可以看到一个弹出菜单。这个菜单显示了点击Speak按钮时所有可能发生的动作。从弹出菜单中的Received Actions区域中选择`speakButtonClicked(_:)`动作。

我们刚创建了一个目标–动作对（target-action pair）。目标–动作对把目标（比如某个类型的实例）和对目标调用的动作（比如一个方法）关联起来，通常用来响应用户的动作（比如点击按钮）。当用户点击Speak按钮时，就会对`ViewController`调用`speakButtonClicked(_:)`方法。

重复这个过程，为Stop按钮创建目标–动作对。按住Control键，然后拖动Stop按钮到`ViewController`图标上，在弹出菜单中选择`stopButtonClicked(_:)`动作。

构建并运行应用，试着点击按钮。回到Xcode，你会看到无论点击哪个按钮都会输出日志信息。这是本章开头在`speakButtonClicked(_:)`和`stopButtonClicked(_:)`方法里所写`print()`调用的结果。

26.5.2　连接文本视图出口

我们已经连接了两个按钮。无论用户什么时候点击它们，对应的方法都会被调用。不过，我们还是拿不到用户在文本视图中输入的文本。要做到这一点，需要连接前面创建的`@IBOutlet`和文本视图。

要连接出口，按住Control键并拖动视图控制器图标到文本视图上，如图26-20所示。注意，拖动顺序跟前面连接按钮动作的顺序相反。

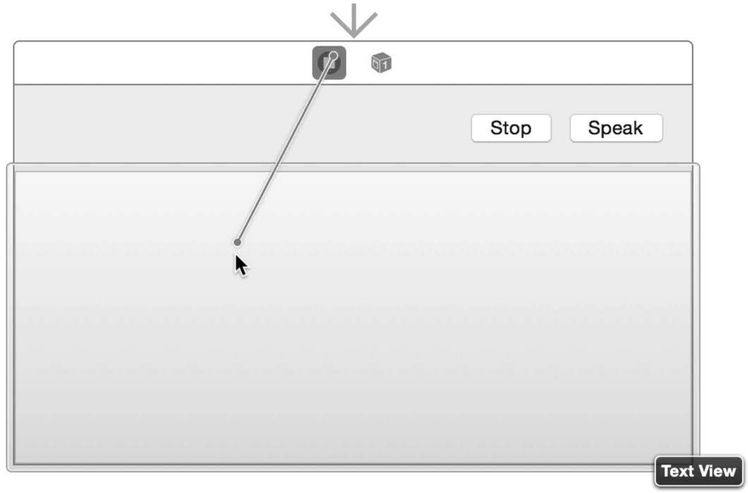


图26-20 连接文本视图出口

在弹出菜单中选择`textView`，跟前面创建的`@IBOutlet`属性的名字对应。

现在可以构建并运行应用了，但是没什么变化。只要用户一点击按钮就会触发`@IBAction`，但是连接`@IBOutlet`本身并不会对程序有什么影响。现在有了到文本视图的出口，就可以打开ViewController.swift并修改`speakButtonClicked(_:)`方法，把当前文本视图的内容打印出来，如代码清单26-3所示。

代码清单26-3 让Speak按钮打印文本视图的内容（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {

    @IBOutlet var textView: NSTextView!

    @IBAction func speakButtonClicked(_ sender: NSButton) {
        print("The speak button was clicked" "I should speak \(textView.string)")
    }

    @IBAction func stopButtonClicked(_ sender: NSButton) {
        print("The stop button was clicked")
    }
}
```

构建并运行应用。在文本视图中输入一些文本，然后点击Speak按钮。你刚才输入的文本会通过`print()`调用打印出来，注意`NSTextView`的`string`属性是可空类型。举个例子，如果在`NSTextView`中输入Hello, world!，你会看到：

```
I should speak Optional("Hello, world!")
```

（如果输入Hello, world!后按下了回车键，你会在控制台看到`\n`，也就是换行符。）

26.6　让 VocalTextEdit“说话”

能打印出文本是很好，但是VocalTextEdit的真正目标是让电脑朗读用户的文本。Cocoa提供了一个NSSpeechSynthesizer类来合成语音。先给ViewController添加一个属性，类型是NSSpeechSynthesizer，如代码清单26-4所示。

代码清单26-4　添加NSSpeechSynthesizer实例（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {

    let speechSynthesizer = NSSpeechSynthesizer()

    @IBOutlet var textView: NSTextView!

    @IBAction func speakButtonClicked(_ sender: NSButton) {
        print("I should speak \(textView.string)")
    }

    @IBAction func stopButtonClicked(_ sender: NSButton) {
        print("The stop button was clicked")
    }
}
```

NSSpeechSynthesizer的默认初始化方法会创建一个使用默认声音的语音合成器。有了语音合成器，就可以修改speakButtonClicked(_:)，让它合成textView的内容了。使用NSSpeechSynthesizer的startSpeaking(_:)方法，它的参数是字符串，如代码清单26-5所示。

代码清单26-5　启动语音合成器（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {

    let speechSynthesizer = NSSpeechSynthesizer()

    @IBOutlet var textView: NSTextView!

    @IBAction func speakButtonClicked(_ sender: NSButton) {
        print("I should speak \(textView.string)")
        if let contents = textView.string {
            speechSynthesizer.startSpeaking(contents)
        } else {
            speechSynthesizer.startSpeaking("The document is empty.")
        }
    }

    @IBAction func stopButtonClicked(_ sender: NSButton) {
        print("The stop button was clicked")
    }
}
```

我们利用可空实例绑定来获取文本视图的内容。如果有内容的话，就朗读出来。如果TextView.string为空，speechSynthesizer会朗读"The document is empty."这句话。

构建并运行应用。输入一些文本（比如Hello, world!），然后点击Speak按钮，你就能听到电脑开始朗读文本了！（确保电脑没有静音。）

现在试着把文本都删掉然后点击Speak。什么都没有发生。为什么电脑没有朗读"The document is empty."呢？

文本视图的内容有两种“空”的形式。第一，string属性是nil。我们写的代码能处理这种情况。第二，string属性不是nil，但是这个字符串包含的是""，即一个空字符串。修改speakButtonClicked(_:)，使其可以处理这种情况，如代码清单26-6所示。

代码清单26-6 处理两种空字符串（ViewController.swift）

```
...
@IBAction func speakButtonClicked(_ sender: NSButton) {
    if let contents = textView.string, !contents.isEmpty {
        speechSynthesizer.startSpeaking(contents)
    } else {
        speechSynthesizer.startSpeaking("The document is empty.")
    }
}
...
```

再次构建并运行应用，把文本视图留空，点击Speak按钮。现在应该能听到合成语音说文档是空的了。

既然VocalTextEdit可以朗读文档了，那么也要能停止朗读。我们已经有了Stop按钮，要做的事情只是修改stopButtonClicked(_:)方法，让它不要调用print()而是做些实际的事情。NSSpeechSynthesizer提供了一个方便的方法来做到我们想做的事，参见代码清单26-7。

代码清单26-7 停下（ViewController.swift）

```
...
@IBAction func stopButtonClicked(_ sender: NSButton) {
    print("The stop button was clicked")
    speechSynthesizer.stopSpeaking()
}
...
```

现在，如果VocalTextEdit正在朗读，那么Stop按钮会马上停止语音。如果VocalTextEdit不在朗读，调用speechSynthesizer.stopSpeaking()也没什么坏处，所以不需要处理这种情况。

构建并运行VocalTextEdit。输入一些文本；文本要足够长，让电脑花一段时间才能读完。然后，点击Speak按钮。当电脑开始朗读后，点击Stop按钮。语音会立即停止。

26.7 保存和加载文档

VocalTextEdit的功能开始变得完整了！我们可以输入文本，听它朗读出来。不过，还缺少一

个很重要的功能：保存文件。你应该已经见过一个文档无法自动保存的通知，如图26-21所示。

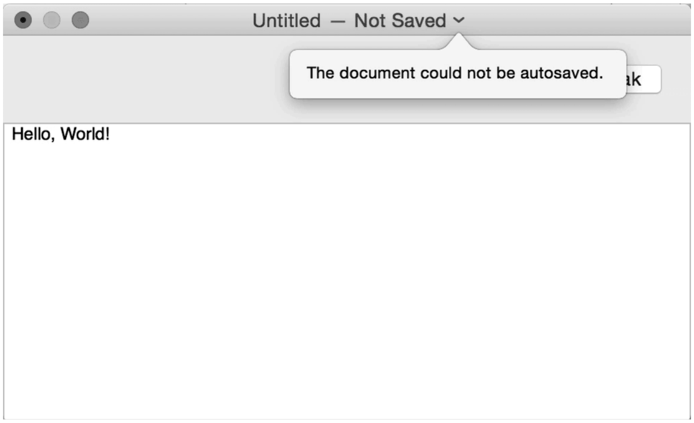


图26-21　VocalTextEdit自动保存失败

更严重的是，如果试图保存文档，会出现一个很讨厌的错误，如图26-22所示。

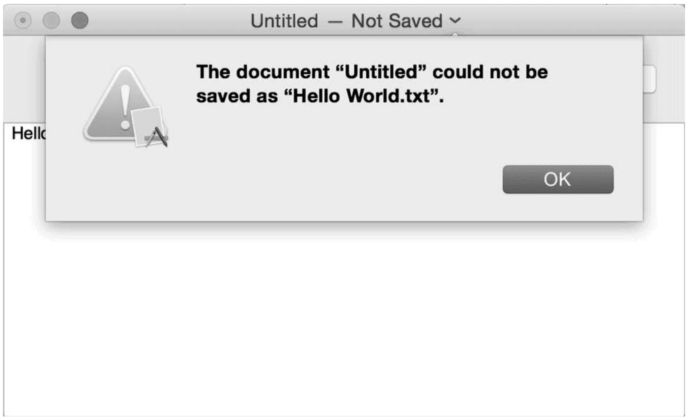


图26-22　VocalTextEdit保存失败

这可能有点奇怪，因为VocalTextEdit就是处理文本文档的。问题在于，虽然你和用户都知道VocalTextEdit是处理文本文档的，但是Cocoa不知道。我们需要补充几个方法让Cocoa保存和加载文档。

先打开Document.swift，删除一些不需要的样板代码，如代码清单26-8所示。

代码清单26-8　清扫代码（Document.swift）

```
import Cocoa

class Document: NSDocument {
```

```
    override init() {
        super.init()
        // Add your subclass specific initialization here.
    }

    override class func autosavesInPlace() -> Bool {
        return true
    }

    override func makeWindowControllers() {
        // Returns the Storyboard that contains your Document window.
        let storyboard = NSStoryboard(name: "Main", bundle: nil)
        let windowController =
            storyboard.instantiateController(
              withIdentifier: "Document Window Controller"
            ) as! NSWindowController

        self.addWindowController(windowController)
    }

    override func data(ofType typeName: String) throws -> Data {
        // Insert code here to write your document to data of the specified type. ...
        throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
    }

    override func read(from data: Data, ofType typeName: String) throws {
        // Insert code here to read your document
        // from the given data of the specified type. ...
        throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
    }
}
```

来看一下其余的方法。

`autosavesInPlace()`是`NSDocument`的类方法。如果`autosavesInPlace()`返回`true`，就表示文档类支持用户随时输入、随时自动保存。因为默认实现返回`false`，所以Xcode很贴心地提供模版代码覆盖原方法并返回`true`，而我们得确保自动保存能正常工作。（这对于VocalTextEdit来说不难。）

26

下一个方法`makeWindowControllers()`会在创建新文档或打开旧文档时被调用。它负责设置`NSWindowController`，这是一个管理文档窗口的类。因为我们用的是故事板，所以设置窗口控制器很容易：从故事板加载窗口控制器（前两行代码），然后添加到文档上（最后一行）。

在`makeWindowControllers()`中有个我们之前没有介绍过的语言特性。`as!`是Swift的类型转换操作符（type casting operator）。

26.7.1　类型转换

类型转换就是告诉编译器："虽然你认为这个对象是X类型，但它实际上是Y类型。"如果是纯Swift代码，一般来说可以避免类型转换，因为继承和泛型往往可以解决问题。不过，在跟用

Objective-C写的库打交道时，要经常用到类型转换。

as!操作符类似于展开一个可空实例。如果试图把一个类型转换成另一个不匹配的类型，就会触发陷阱。（如果不记得什么是陷阱，回去看看第20章。）类型转换操作符还有两种变体：as?会试图进行类型转换，如果不匹配则返回nil；而as会进行Swift编译器保证一定成功的转换，比如从NSString到String。

按住Option键，点击在makeWindowControllers()第二行调用的instantiateController(withIdentifier:)方法。注意，它的返回值是Any。Any是一个Swift协议，没有方法也没有属性：它可以表示任何可能的类型。instantiateController(withIdentifier:)返回Any是因为故事板可能包含很多不同类型的控制器。为了利用返回的控制器，就需要把它转换成实际类型：NSWindowController。在创建一个基于故事板的应用时，Xcode会自动为创建的NSWindow-Controller设置标识Document Window Controller，并在这一行代码中自动插入同一个标识。

26.7.2　保存文档

Document.swift中最后两个方法是用来支持保存和加载的。当需要保存文档时，data(ofType:)方法就会被调用。它一般会返回一个Data实例。你可以把Data理解为字节数组。如果想保存文档， data(ofType:)需要返回要保存的字节。如果因为任何原因无法保存文档，就抛出一个错误。

与之关联的实现加载文档的方法是read(from:ofType:)。它的第一个参数是一个Data实例，而read(from:ofType:)就是从这些字节中加载文档的。如果加载失败，就抛出错误。

目前，data(ofType:)和read(from:ofType:)都会失败，这解释了为什么我们会看到自动保存失败。注意，方法中有些注释会告诉你需要做什么：// Insert code here to write your document to data of the specified type.。现在根据注释的提示来修复这些问题。

首先实现data(ofType:)，使它能保存VocalTextEdit文本文件。要保存文档，需要把文本视图的内容转化成Data。第一步是得到跟Document关联的ViewController。注意，我们在代码清单26-9中省略了Xcode插入的模版注释，这样可以节约点纸张。

代码清单26-9　得到关联的ViewController（Document.swift）

```
...
override func data(ofType typeName: String) throws -> Data {
    // Insert code here to write your document to data of the specified type. ...
    throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
    let windowController = windowControllers[0]
    let viewController = windowController.contentViewController as! ViewController
}
...
```

data(ofType:)还没有完成，编译器还在抱怨我们没有返回值。修复这个问题之前，先来看看我们添加的代码。

一般来说，NSDocument可能有多个窗口控制器。在本例中，我们只创建了一个（在

makeWindowControllers()中)，所以windowControllers[0]会让我们得到唯一的窗口控制器。

一个NSWindowController可能有一个视图控制器作为内容，也可能没有——Mac平台有窗口控制器的历史比有视图控制器的历史久多了。按住Option键并点击contentViewController，你会看到其类型是NSViewController?。窗口控制器可能没有内容视图控制器，此时这个属性就是nil。如果windowController有内容视图控制器，编译器就会知道它要么是NSViewController，要么是NSViewController的子类。

在这个应用中，只有一种ViewController类型；不过对于大型应用，可能会有很多。用as!类型转换操作符可以告诉编译器："我知道Document窗口控制器的视图控制器类型是ViewController。"

有了视图控制器，就可以实现data(ofType:)的剩余部分了，如代码清单26-10所示。

代码清单26-10 实现data(ofType:)完成保存功能（Document.swift）

```
import Cocoa

class Document: NSDocument {
    enum Error: Swift.Error, LocalizedError {
        case UTF8Encoding

        var failureReason: String? {
            switch self {
            case .UTF8Encoding: return "File cannot be encoded in UTF-8."
            }
        }
    }
    ...
    override func data(ofType typeName: String) throws -> Data {
        let windowController = windowControllers[0]
        let viewController = windowController.contentViewController as! ViewController
        let contents = viewController.textView.string ?? ""

        guard let data = contents.data(using: .utf8) else {
            throw Document.Error.UTF8Encoding
        }
        return data
    }
    ...
```

我们来一点一点地解析这段代码。

```
enum Error: Swift.Error, LocalizedError {
    case UTF8Encoding

    var failureReason: String? {
        switch self {
        case .UTF8Encoding: return "File cannot be encoded in UTF-8."
        }
    }
}
```

首先，跟第20章一样，声明一个符合Swift.Error的错误类型，我们还让它符合了LocalizedError协议。LocalizedError能以对用户更友好的方式显示错误消息。（在实际的应用中，failureReason返回的字符串应该本地化成不同的语言，不过本书不讨论这个话题。）如果保存文件失败，Cocoa会拿到错误的failureReason，并放进展示给用户的警告框中。

```
let contents = viewController.textView.string ?? ""
```

接着尝试通过string属性从文本视图获取内容，方法与在ViewController中获取文本来合成语音一样。如果string是nil，利用第8章中出现过的nil合并运算符把""（即空字符串）作为默认值。

```
guard let data = contents.data(using: .utf8) else {
    throw Document.Error.UTF8Encoding
}

return data
```

最后，对字符串使用data(using:)来尝试把内容转化成Data。data(using:)方法要求我们指定字符串编码——也就是如何把contents里的文本转化成字节。.utf8是枚举String.Encoding的一个成员，表示字符串应该用UTF-8编码，UTF-8是一种常见的Unicode编码。如果转化失败，就抛出上面定义的错误。如果转化成功，则返回新创建的数据。

构建并运行应用。在文档中输入一些文本然后保存。成功了！好吧，是几乎成功了。关闭保存了的文档，然后尝试通过File → Open打开。

加载不出来。我们能保存文档，但是加载文档还需要实现read(from:ofType:)。

26.7.3　加载文档

保存文档的时候，我们从视图控制器的文本视图中获取内容，将字符串转化成Data，然后返回数据。要加载文档，把这个过程反过来似乎是合理的：我们拿到Data的实例，把它转化成字符串，然后把内容放到视图控制器的文本视图中。试一下如代码清单26-11所示的代码。初看好像是对的，但是它实际上有个大问题。

代码清单26-11　加载文档——错误的方式（Document.swift）

```
import Cocoa

class Document: NSDocument {
    enum Error: Swift.Error, LocalizedError {
        case UTF8Encoding
        case UTF8Decoding

        var failureReason: String? {
            switch self {
            case .UTF8Encoding: return "File cannot be encoded in UTF-8."
            case .UTF8Decoding: return "File is not valid UTF-8."
            }
        }
```

```
    }
    ...
    override func read(from data: Data, ofType typeName: String) throws {
        // Insert code here to read your document
        // from the given data of the specified type...
        throw NSError(domain: NSOSStatusErrorDomain, code: unimpErr, userInfo: nil)
        guard let contents = String(data: data, encoding: .utf8) else {
            throw Document.Error.UTF8Decoding
        }

        // 警告：这里有严重问题
        let windowController = windowControllers[0]
        let viewController = windowController.contentViewController
          as! ViewController
        viewController.textView.string = contents
    }
}
```

构建并运行应用。关闭所有窗口，尝试打开几分钟前保存的文档。不好，文档打不开——应用似乎什么都没做。如果查看Xcode的控制台，你会看到一段很长的错误消息，开头是这样的：

```
index 0 beyond bounds for empty NSArray
```

这个错误告诉我们，当我们试图从`windowControllers`数组中获取元素0时，它并不存在。为什么不存在？因为Cocoa在创建窗口以及与之关联的控制器之前调用了`read(from:ofType:)`，所以文档还没有窗口控制器，也没有视图控制器。

因此，这个实现从表面看逻辑正确，实际却不能运行。为了避免这个错误，我们可以保留好字符串，等到视图控制器从故事板中加载出来再更新其内容。首先新建一个属性来保存内容，在视图控制器能用之后用这部分内容填充，如代码清单26-12所示。

代码清单26-12 用更好的方式加载文档——创建contents属性（Document.swift）

```
import Cocoa

class Document: NSDocument {
    ...
    var contents: String = ""

    override class func autosavesInPlace() -> Bool {
        return true
    }
    ...
}
```

我们可以把`contents`的默认值设置为空字符串，因为对于不是从现有文件打开的新文档来说，这就是正确的值。接下来，修改`read(from:ofType:)`，把数据存入这个属性，而不是更新还不存在的视图控制器，如代码清单26-13所示。

代码清单26-13 用更好的方式加载文档——把内容存入contents属性（Document.swift）

```
    ...
    override func read(from data: Data, ofType typeName: String) throws {
```

```
        guard let contents = String(data: data, encoding: .utf8) else {
            throw Document.Error.UTF8Decoding
        }

        // 警告：这里有严重问题
        let windowController = windowControllers[0]
        let viewController = windowController.contentViewController
          as! ViewController
        viewController.textView.string = contents
        self.contents = contents
    }
}
```

最后，当创建视图控制器之后，要把文档内容传递给它。更新makeWindowControllers()实现这个功能，如代码清单26-14所示。

代码清单26-14　用更好的方式加载文档——把文档内容传递给视图控制器（Document.swift）

```
...
override func makeWindowControllers() {
    // Returns the Storyboard that contains your Document window.
    let storyboard = NSStoryboard(name: "Main", bundle: nil)
    let windowController =
        storyboard.instantiateController (
          withIdentifier: "Document Window Controller"
        ) as! NSWindowController

    let viewController = windowController.contentViewController as! ViewController
    viewController.textView.string = contents

    self.addWindowController(windowController)
}
...
```

构建并运行应用，现在可以成功打开文件了！VocalTextEdit快完成了。

26.7.4　按照 MVC 模式整理代码

VocalTextEdit的功能已经完成了：可以保存和加载文档，还可以朗读文档。但是在宣布全部完成之前，回想一下我们之前说过的MVC："模型对用户界面一无所知。"我们把Document用作模型类，但是它在好几个地方都染指了文本视图。文本视图绝对是用户界面的一部分！

ViewController应该是用户界面（文本视图）和模型（文档）之间的桥梁。从良好的编程风格角度考虑，我们需要整理VocalTextEdit的分层。首先打开ViewController.swift，为文本视图的内容新建一个属性，如代码清单26-15所示。

代码清单26-15　为文本视图的内容新建属性（ViewController.swift）

```
import Cocoa

class ViewController: NSViewController {
```

```
    let speechSynthesizer = NSSpeechSynthesizer()

    @IBOutlet var textView: NSTextView!

    var contents: String? {
        get {
            return textView.string
        }
        set {
            textView.string = newValue
        }
    }
    ...
}
```

这里新建了一个计算属性contents，其读取方法会从textView的string属性读取内容，而写入方法会往这个属性写入内容。

现在，回到Document.swift，用这个新属性替换对视图控制器的文本视图的访问，如代码清单26-16所示。

代码清单26-16 用contents替换对文本视图的引用（Document.swift）

```
    ...
    override func makeWindowControllers() {
        // Returns the Storyboard that contains your Document window.
        let storyboard = NSStoryboard(name: "Main", bundle: nil)
        let windowController =
            storyboard.instantiateController(
              withIdentifier: "Document Window Controller"
            ) as! NSWindowController

        viewController = windowController.contentViewController as! ViewController
        viewController.~~textView.string~~contents = contents

        self.addWindowController(windowController)
    }

    override func data(ofType typeName: String) throws -> Data {
        let windowController = windowControllers[0]
        let viewController = windowController.contentViewController as! ViewController
        let contents = viewController.~~textView.string~~contents ?? ""

        guard let data = contents.data(using: .utf8) else {
            throw Document.Error.UTF8Encoding
        }

        return data
    }
    ...
```


构建并运行VocalTextEdit，确认我们还可以保存和加载文件。

刚才对代码的重构看上去很小，但是很重要。Document不能也不应该关心字符串如何显示。

ViewController类负责管理用户界面（所以它知道文本视图）以及与文档通信（所以它向文档暴露了文档所需的接口：contents属性）。

这个变化不仅仅是一个良好的编程实践，还有另外两个好处。第一，Document的可读性更好，因为代码更直接地表达了意图。为了保存文档，从视图控制器获取内容并保存。为了加载文档，把内容放进属性；一旦视图控制器准备好，就把内容传递给它。

第二，重构后的代码能让Document更好地应对将来视图控制器可能发生的变化。举个例子，你可能想再添加一个文本视图让用户输入文档标题，或者想用一个跟文本视图不同的视图。既然明确了Document和ViewController的交互关系，那么这种改变就不大会影响Document的可读性和正确性。

26.7.5　现实世界中的 Swift

恭喜你，你完成了一个Mac应用！这个应用可能并不太复杂，但这不是重点。重点是你应用了Swift知识，也使用了一些苹果提供的Cocoa库，还写了一个功能完善、设计良好的应用。太棒了！

作为一名程序员，在阅读本章的过程中，你的第六感可能告诉你某些技术不太对。隐式展开可空实例可能很危险，as!操作符也可能很危险，而这两种技术在本章中得到了多次使用。

遗憾的是，一旦开始和为Objective-C设计的工具和框架交互，就不能保证总是能实现Swift的纯粹性和安全性了。有些妥协不可避免，尤其是和围绕Interface Builder构建的系统交互时。不过还是振作点，在大型应用中，一部分代码的危险性能被应用中其他部分使用Swift所带来的安全性所抵消。

26.8　白银挑战练习

目前，VocalTextEdit的用户可以在任何时候点击Speak或Stop按钮。这不太理想。举个例子，在VocalTextEdit正在播放合成语音时点击Speak会突然重新播放语音。修改VocalTextEdit，只有在不播放语音时才能点击Speak，只有在播放语音时才能点击Stop。

要完成这个练习，需要设置两个按钮的enabled属性，还需要知道播放什么时候结束。（什么时候开始我们已经知道了。）查看NSSpeechSynthesizerDelegate协议的文档可以找到这些信息。

26.9　黄金挑战练习

如果还没有完成白银挑战练习，先完成再回来！

现在，当应用开始或停止朗读时按钮会有响应，接下来让用户知道朗读会持续多久。在界面上添加NSProgressIndicator，更新这个控件以显示已经完成部分的估计值。作为一个额外练习，让应用只在朗读时才显示进度条。

第 27 章

第一个iOS应用

在本章中，我们会为iPhone创建一个名为iTahDoodle的iOS应用。iTahDoodle允许用户创建待办事项列表。跟VocalTextEdit一样，iTahDoodle是一个比较简单的应用，构建这个应用只能让你稍微体验一下iOS开发。要深入学习iOS编程，请参考最新版的《iOS编程》。

完成后，iTahDoodle看起来如图27-1所示。

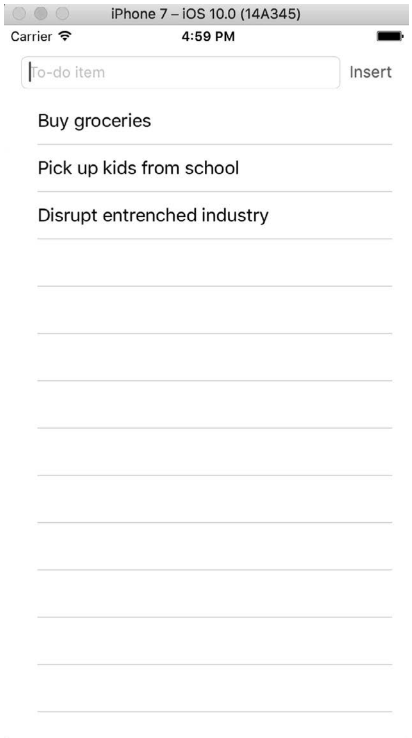


图27-1　完成后的iTahDoodle应用

用户可以在顶部的文本框中输入待办事项并点击Insert将其添加到列表中。我们需要让待办事项列表持续存在，这样即使用户关闭应用也不会丢失列表。

27.1　开始创建 iTahDoodle

在Xcode中，选择File → New → Project...。选择中顶部的iOS区域。然后选择Single View模版（如图27-2所示）并点击Next。

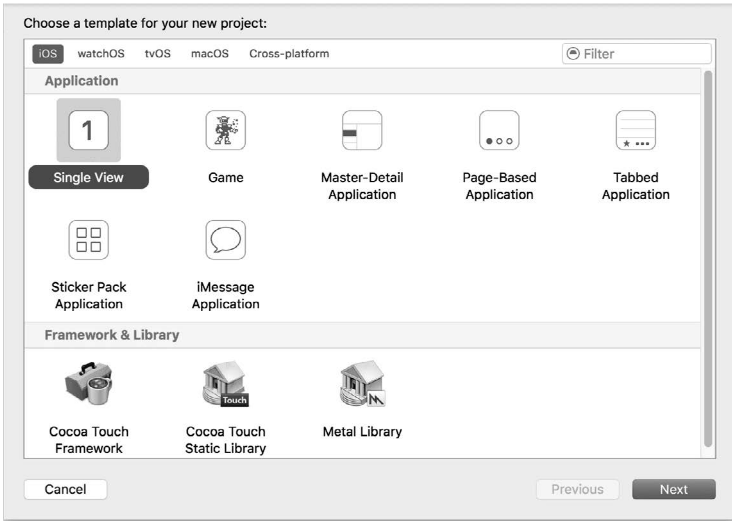


图27-2　选择iOS的单视图应用模版

在工程选项窗口将工程命名为iTahDoodle，如图27-3所示。确保选择语言是Swift，设备是iPhone。确保不选中Use Core Data、Include Unit Tests和Include UI Tests。

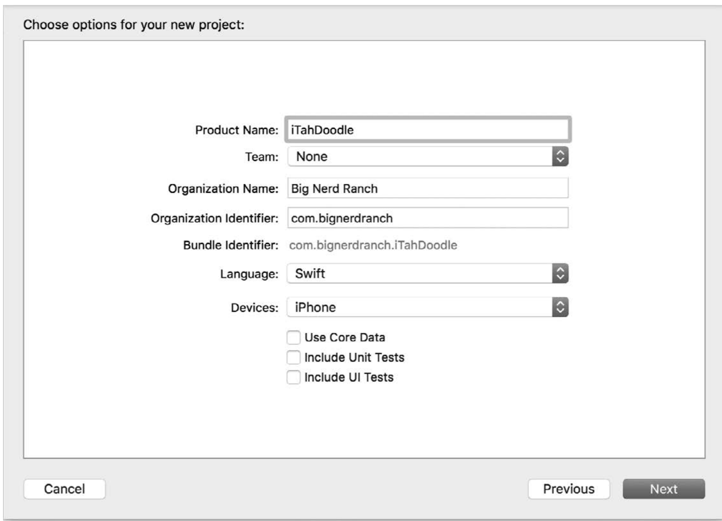


图27-3　配置iTahDoodle

点击Next，把它保存到你想要的地方，完成工程的创建。

27.2　布局用户界面

从工程导航区选择Main.storyboard。iOS的单视图应用模版比上一章中用到的Cocoa故事板简单多了。现在，故事板只包含一个空的视图控制器。

为视图控制器添加一个按钮。在对象库(位于工具面板的底部)中搜索button。拖动一个Button到视图控制器画布上，放在右上角时它会吸附到蓝色虚线上。最后，在属性检查面板（attributes inspector）中把按钮的标题改为Insert。现在视图控制器看起来应该类似于图27-4。

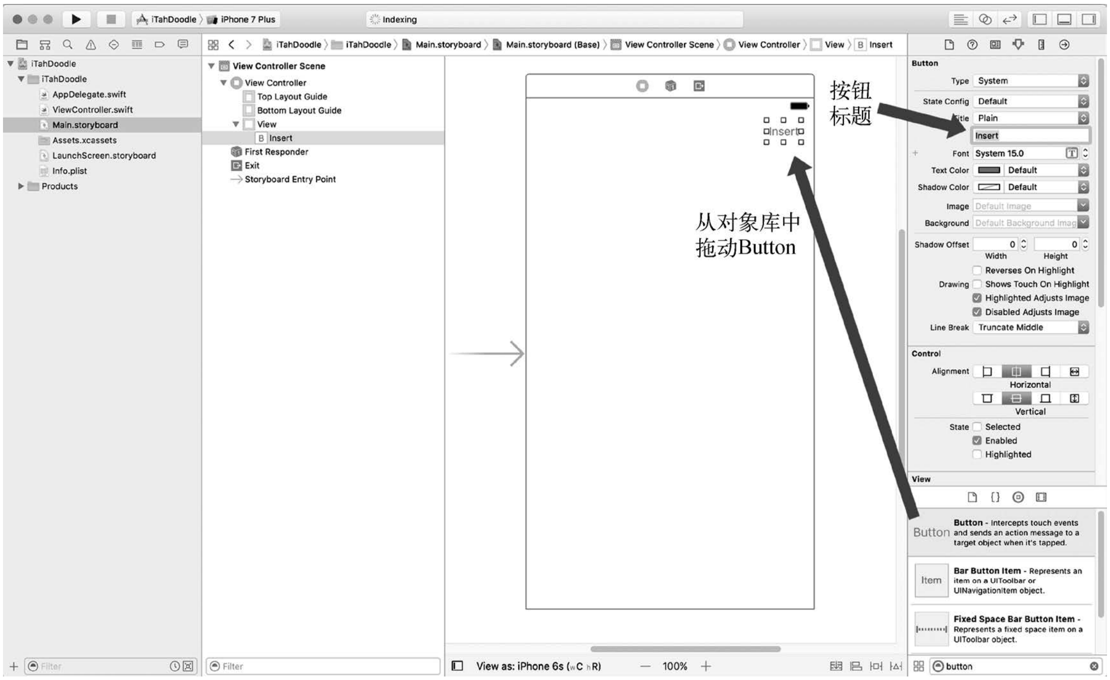


图27-4　有Insert按钮的iTahDoodle

接着，添加一个文本框，以便用户输入待办事项。在对象库中搜索text field。把一个Text Field拖动到视图控制器画布的左上角。用方形的缩放锚点调整文本框的大小，使得其右边缘和刚才添加的按钮接触。用属性检查面板把文本框的占位文本设置为To-do Item（如图27-5所示）。

在上一章中，我们使用自动布局来确保用户在缩放窗口时用户界面看起来是正常的。目前，用户还无法缩放iOS应用的窗口，不过也有一个类似的问题：我们希望应用在不同屏幕尺寸下看起来是正常的。指定iPhone为目标设备（在设置iTahDoodle的时候指定的）还不够，因为有很多屏幕尺寸各不相同的iPhone机型。不过，自动布局系统可以确保界面根据屏幕尺寸自动调整大小。

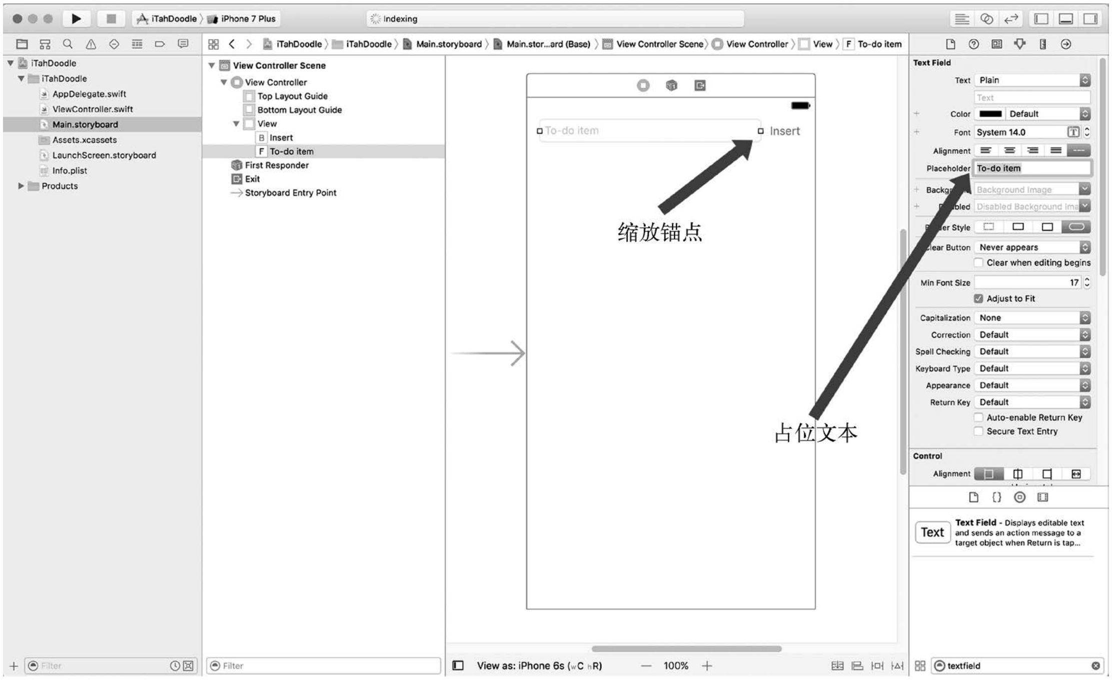


图27-5　有文本框的iTahDoodle

在文档大纲中，按住Control键从Insert拖动到其父视图。在弹出菜单中，按住Shift键并点击Trailing Space to Container Margin和Vertical Spacing to Top Layout Guide（如图27-6所示）。

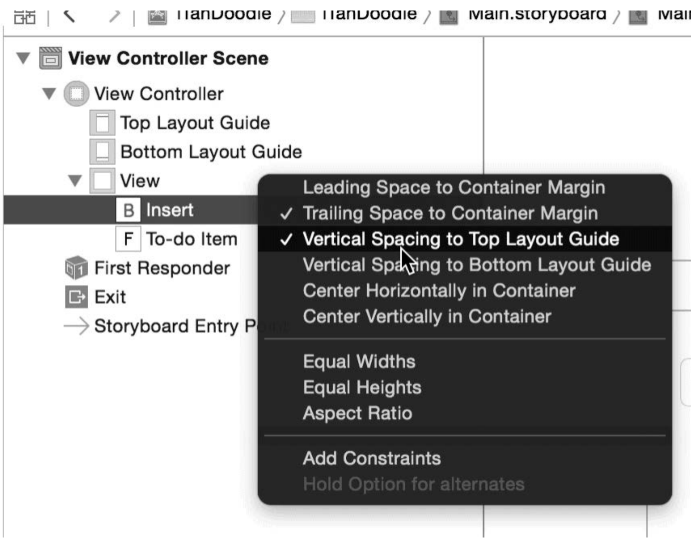


图27-6　为Insert按钮添加自动布局约束

这些约束会确保按钮固定在视图的右上角。

接下来，在文本框和按钮之间创建约束。按住Control键从文本框拖动到按钮。在弹出菜单中，按住Shift键并点击Horizontal Spacing和Baseline（如图27-7所示）。（你可能已经注意到Interface Builder中有一个表示警告的黄色图标——我们将很快解决。）

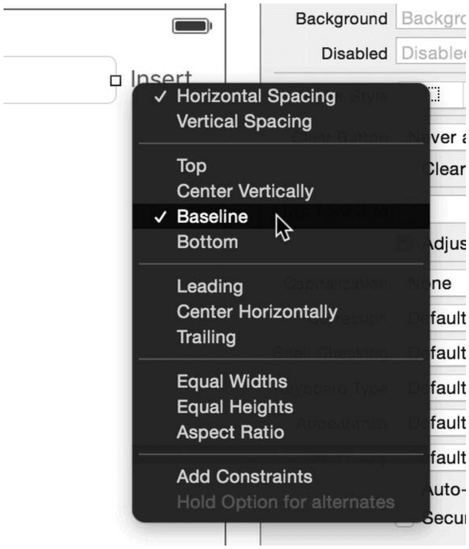


图27-7　在文本框和按钮之间添加自动布局约束

这些约束确保文本框的右边紧贴按钮，而且文本框中的文本和按钮上的文本垂直对齐。

最后要添加的约束把文本框的左边缘和视图的左边缘对齐。在文档大纲中，按住Control键从文本框拖动到父视图，在弹出菜单中选择Leading Space to Container Margin（如图27-8所示）。

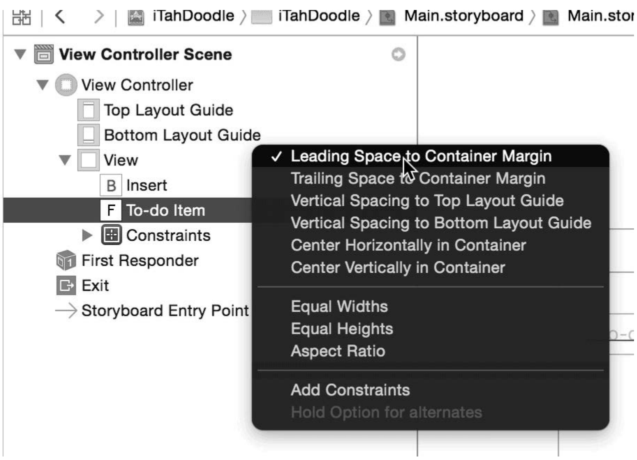


图27-8　在文本框和父视图之间添加自动布局约束

27

你可能注意到Interface Builder还是有警告。按住Command-4组合键打开问题导航器，你会看到2 views are horizontally ambiguous。问题在于我们约束文本框和按钮撑满整个水平空间，但是自动布局不知道如何分配空间。根据刚才添加的约束，自动布局可以让按钮很窄而文本框很宽，也可以反过来（或者任何一种中间状态）。要修复这个问题，选择Insert按钮，在工具面板中打开尺寸检查面板，把Horizontal Content Hugging Priority从250改成251（如图27-9所示）。

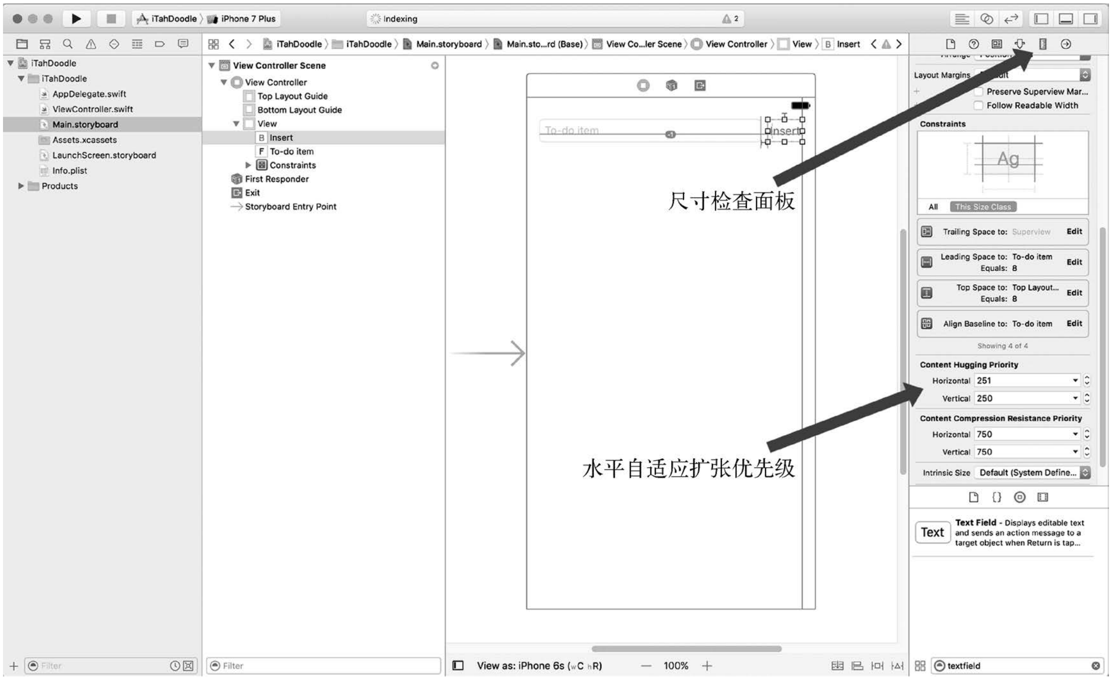


图27-9　Insert按钮的水平自适应扩张优先级

自适应扩张优先级（content hugging priority）决定自动布局多强烈地阻止元素扩张。我们把按钮的自适应扩张优先级增加到251，大于文本框的默认值250。当自动布局视图判断如何填充整个水平空间时，它会看到按钮不想水平扩张。因此，文本框会占满所有可用的空间，而按钮尺寸保持不变。

这里可能会出现一两个misplaced views警告，意味着现在在故事板中看到的视图边框和应用运行起来并且自动布局接管之后的实际位置不一样。Xcode有一系列工具解决类似于这样的布局问题。最常用的工具是更新视图的边框使它和自动布局认为的一样。现在点击Resolve Auto Layout Issues按钮，选择All Views in View Controller区域中的Update Frames来使用这个工具（如图27-10所示）。

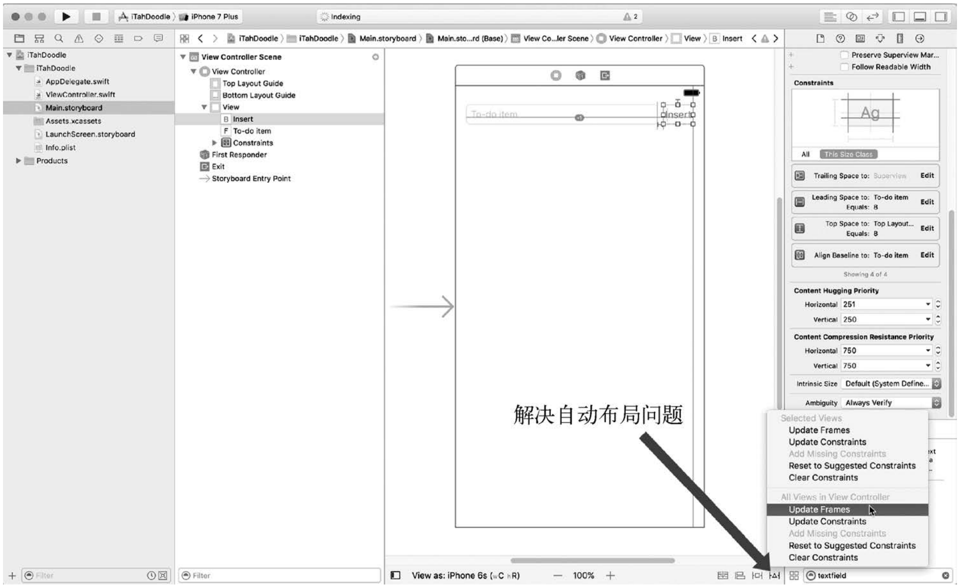


图27-10　解决自动布局问题：Update Frames

我们需要的最后一个UI元素用于显示待办事项列表。iOS提供的`UITableView`类可以完美实现这个目标。在对象库中搜索table，拖动Table View到视图控制器上（确认拖动的是Table View而不是Table View Controller），如图27-11所示。

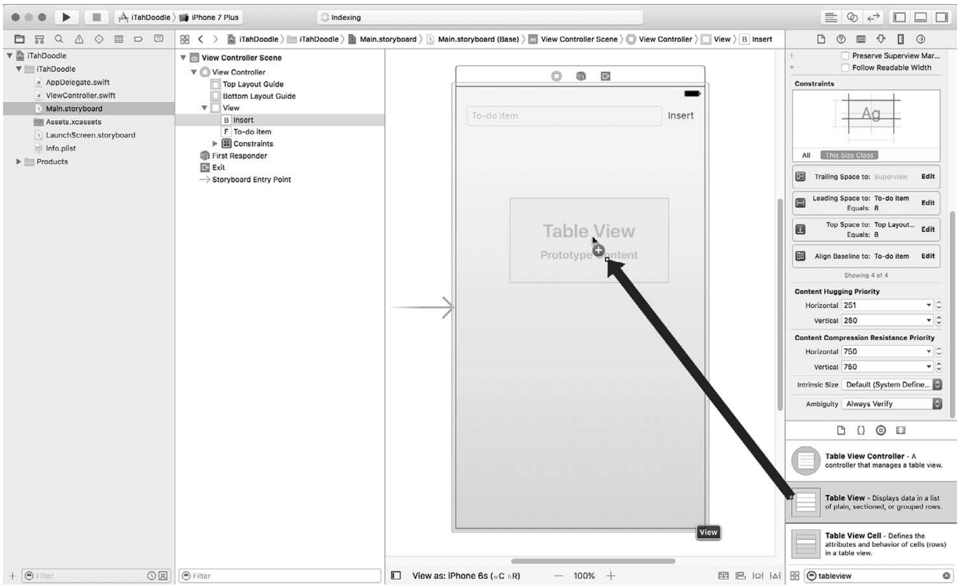


图27-11　添加表格视图

调整表格视图的大小直到紧贴四周的边缘，如图27-12所示。

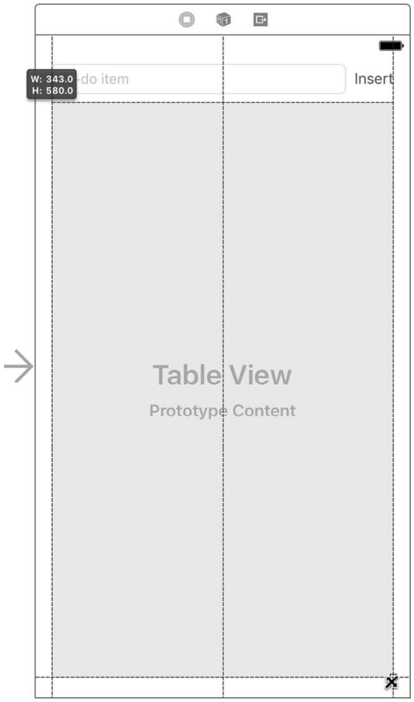


图27-12　调整表格视图的大小，让它占满视图控制器

我们需要添加自动布局约束来确保表格视图总是占满可用的屏幕空间。在文档大纲中，按住Control键从表格视图拖动到其父视图。在弹出菜单中，按住Shift键并点击Leading Space to Container Margin、Trailing Space to Container Margin和Vertical Spacing to Bottom Layout Guide（如图27-13所示）。

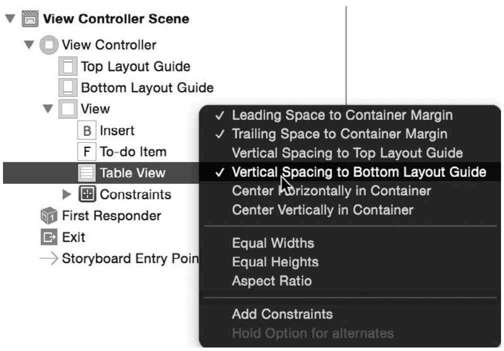


图27-13　表格视图和父视图之间的自动布局约束

最后，添加一个约束来修复表格视图和上面文本框之间的距离问题。按住Control键从表格视图拖动到文本框，在弹出菜单中选择Vertical Spacing（如图27-14所示）。

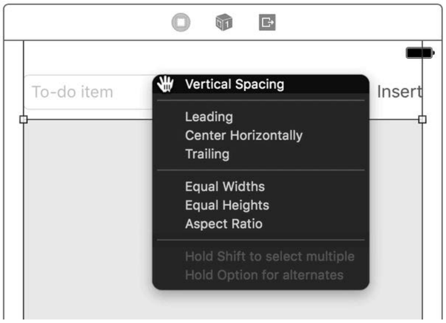


图27-14　表格视图和文本框之间的自动布局约束

构建并运行应用。这样会在Xcode的iOS模拟器中打开iTahDoodle，如图27-15所示。它现在还没有什么功能，不过已经能看到我们设计的用户界面了。

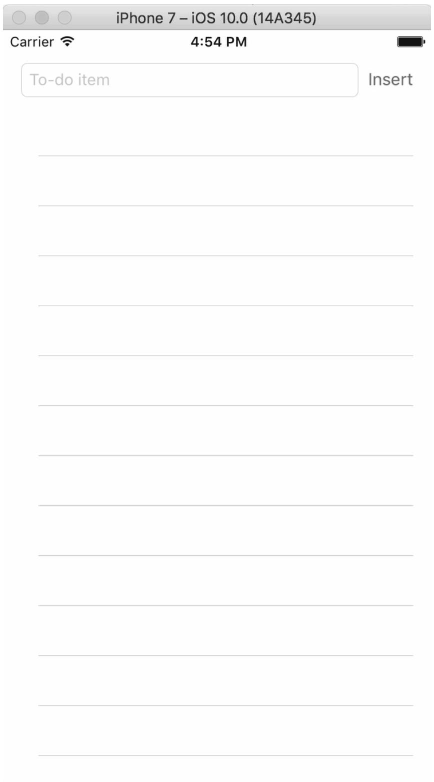


图27-15　iTahDoodle的用户界面

连接用户界面

既然有了界面，就可以创建出口和动作了。这样可以让应用响应交互。打开ViewController.swift，为UI元素添加属性和用户点击Insert按钮时要调用的动作方法，如代码清单27-1所示。

代码清单27-1　添加UI元素属性和按钮动作方法（ViewController.swift）

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var itemTextField: UITextField!
    @IBOutlet var tableView: UITableView!

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func addButtonPressed(_ sender: UIButton) {
        print("Add to-do item: \(itemTextField.text)")
    }
}
```

接着，回到Main.storyboard，连接刚才添加的代码。按住Control键从视图控制器拖动到文本框，从弹出菜单中选择itemTextField（如图27-16所示）。

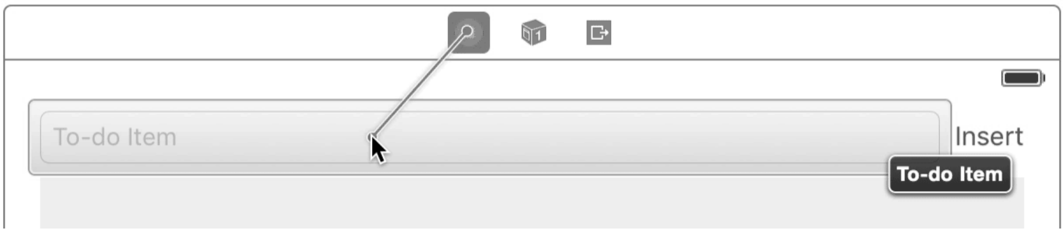


图27-16　把文本框连接到视图控制器的@IBOutlet

重复这个过程，连接表格视图。按住Control键从视图控制器拖动到表格视图，在弹出菜单中选择tableView。

我们在第26章讲过，连接一个动作的时候，要按住Control键并向相反的方向拖动。按住Control键从Insert按钮拖动到视图控制器（如图27-17所示），在弹出菜单中的Sent Events下面选择addButtonPressed:。

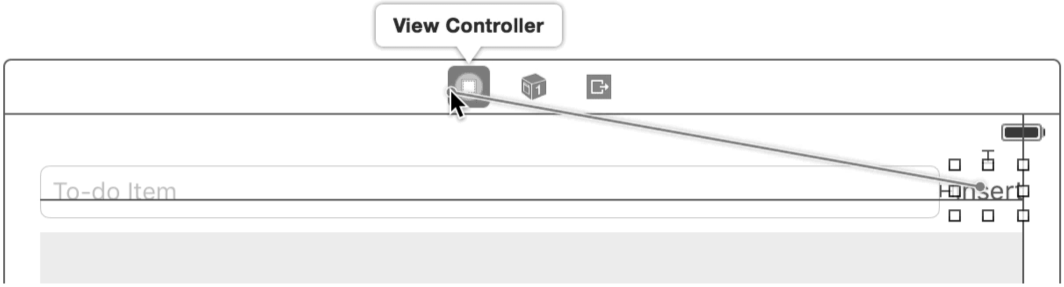


图27-17 把Insert按钮连接到视图控制器的`@IBAction`

再次构建并运行应用。试着在文本框中输入一些文本再点击Insert按钮。文本会在控制台中打印出来。举个例子，如果输入Buy groceries，点击Insert，然后输入Walk the dog，再点击Insert，就会看到如下输出：

```
Add to-do item: Optional("Buy groceries")
Add to-do item: Optional("Walk the dog")
```

27.3 为待办事项列表建模

还记得第26章讲过的MVC架构吗？我们已经创建了视图（故事板和UI元素）和控制器（`ViewController`类），不过现在还没有模型。选择File → New → File...创建一个Cocoa Touch类。选中顶部的iOS，在Source区域选择Cocoa Touch Class（如图27-18所示）。

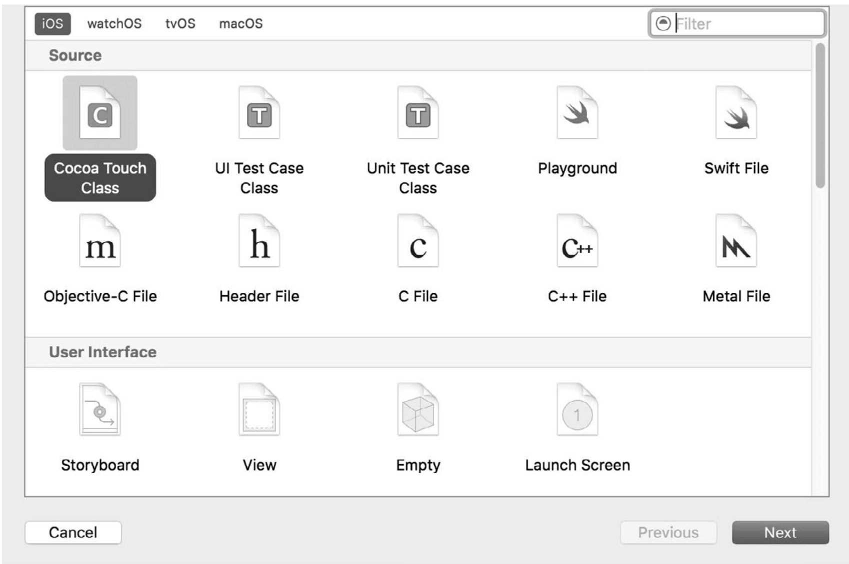


图27-18 创建新的Cocoa Touch类

在下一屏把类命名为TodoList，让它继承NSObject，确保语言是Swift（如图27-19所示）。

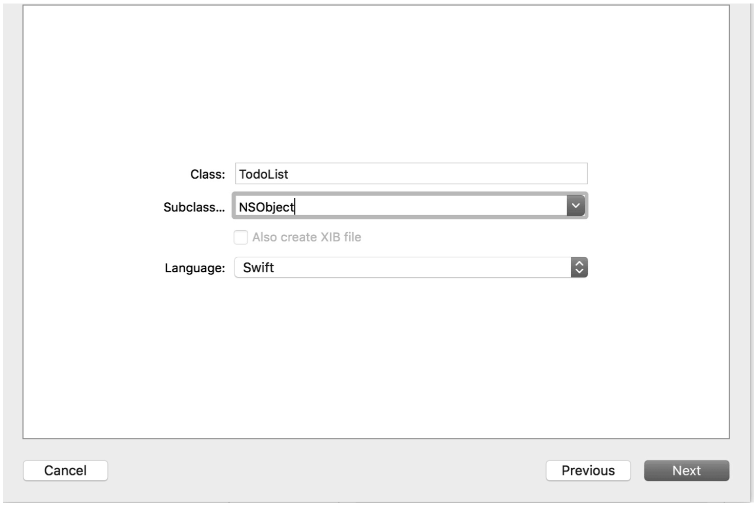


图27-19　创建TodoList类

点击Next，然后点击Create。

Xcode会创建并打开新类：

```
import UIKit

class TodoList: NSObject {

}
```

NSObject是什么？在Swift中，我们可以创建没有父类的类。在Objective-C中，所有的类都需要有父类。NSObject被称为根类（root class）：它能提供一些基本的Objective-C运行时支持。

我们需要让TodoList继承NSObject是因为要用它和Cocoa Touch类打交道，而Cocoa Touch类期望接受Objective-C对象。（Swift和Objective-C的关系是第28章的主题。）

从最基本的层面而言，待办事项列表只是一个能往里添加东西的字符串列表。为TodoList添加属性和方法来满足这些条件，如代码清单27-2所示。

代码清单27-2　添加基本的列表功能（TodoList.swift）

```
import UIKit

class TodoList: NSObject {
    fileprivate var items: [String] = []

    func add(_ item: String) {
        items.append(item)
    }
}
```

我们之前把一个UITableViwe放到界面上来展示待办事项列表。每个UITableView都有一个dataSource属性提供表格单元的内容。想要被用作表格视图的数据源，TodoList必须符合UITableViewDataSource协议。正如我们在第21章学到的，可以把符合协议的代码添加到扩展中，以便把相关的功能组合在一起。在TodoList上添加一个扩展，使它符合UITableViewDataSource，如代码清单27-3所示。

代码清单27-3　在扩展中添加符合协议的声明（TodoList.swift）

```
import UIKit

class TodoList: NSObject {
    fileprivate var items: [String] = []

    func add(_ item: String) {
        items.append(item)
    }
}

extension TodoList: UITableViewDataSource {
}
```

如果现在试图构建工程，会发现一个TodoList不符合UITableViewDataSource的错误。打开UITableViewDataSource的文档（如图27-20所示）。（按住Option键并点击UITableViewDataSource打开快速参考文档，然后点击UITableViewDataSource Protocol Reference链接打开完整的文档。）

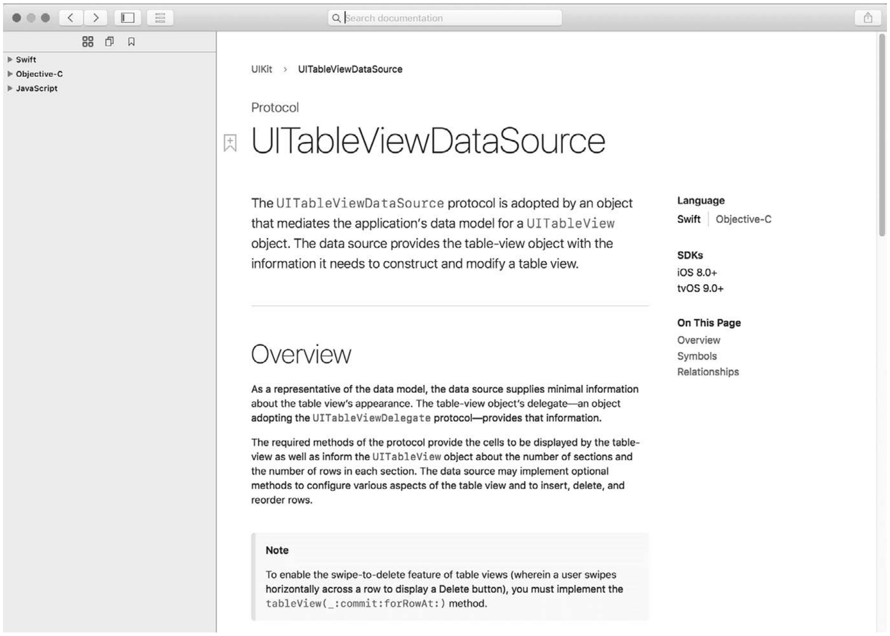


图27-20　UITableViewDataSource协议参考文档

UITableViewDataSource中有大量方法！不过，只有两个是必需的：负责配置并返回表格单元（表格视图的行）的tableView(_:cellForRowAtIndexPath:)和负责告诉表格视图有多少行的tableView(_:numberOfRowsInSection:)。首先实现tableView(_:numberOfRowsInSection:)，如代码清单27-4所示。

代码清单27-4　添加tableView(_:numberOfRowsInSection:)（TodoList.swift）

```
...
extension TodoList : UITableViewDataSource {
    func tableView(_ tableView: UITableView,
                   numberOfRowsInSection section: Int) -> Int {
        return items.count
    }
}
```

UITableView支持一个表格有多个表头，每个表头可以有0行或更多行。对iTahDoodle来说，只要用到一个表头，所以上面的方法忽略了section参数。我们返回items.count告诉表格每个待办事项都是一行。

接着实现tableView(_:cellForRowAt:)，如代码清单27-5所示。

代码清单27-5　添加tableView(_:cellForRowAt:)（TodoList.swift）

```
...
extension TodoList : UITableViewDataSource {
    func tableView(_ tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
        return items.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath)
            -> UITableViewCell {
        let cell = tableView.dequeueReusableCell(withIdentifier:
                   "Cell", for: indexPath)
        let item = items[indexPath.row]

        cell.textLabel!.text = item

        return cell
    }
}
```

下面一行一行地对这个方法进行分析。

```
let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
```

这一行让表格视图从队列中取出有"Cell"标识和给定索引路径的可复用表格单元。

可复用的表格单元（reusable cell）是什么意思？为了在移动设备上达到良好的滑动性能，UITableView会使用表格单元的复用池（reuse pool）。当系统不再需要表格单元时（比如随着用户的滑动，表格单元滑出屏幕），表格视图会把表格单元放进复用池。如果表格视图的复用池中有表格单元，就会从池中取出它并返回；否则创建一个新的。无论何种情况，都能确保有一个实

例返回。

先不管"Cell"标识以及它是怎么变成UITableViewCell的，我们稍后会讲到。

```
let item = items[indexPath.row]
```

每次表格视图需要数据源配置一个即将显示给用户的表格单元的时候都会调用tableView(_:cellForRowAt:)。indexPath参数表示表格视图需要显示哪一行。它有表示section和row的属性。上面我们讲过，iTahDoodle的表格视图只有一个表头，所以可以忽略表头，只需要基于row来查看要显示哪个待办事项。

```
cell.textLabel!.text = item
```

现在有了UITableViewCell和要显示的待办事项，这一行把表格单元的textLabel的text属性设置为待办事项。我们强制展开了textLabel；不是所有的UITableViewCell都有textLabel，但是我们用的这个有。

```
return cell
```

最后，返回配置好的表格单元。

现在TodoList可以用作UITableView的数据源了。构建应用，确保我们正确实现了所有方法。现在还看不出变化，因为表格视图还需要设置。但是至少没有错误了。

27.4 设置 UITableView

模型类准备好了。回到ViewController.swift的控制器。添加一个TodoList的实例作为属性，修改addButtonPressed(_:)，把待办事项添加到todoList而不是打印出来，如代码清单27-6所示。

代码清单27-6 给控制器添加模型类作为属性（ViewController.swift）

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var itemTextField: UITextField!
    @IBOutlet var tableView: UITableView!

    let todoList = TodoList()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
```

```
    @IBAction func addButtonPressed(_ sender: UIButton) {
        print("Add to-do item: \(itemTextField.text)")
        guard let todo = itemTextField.text else {
            return
        }
        todoList.add(todo)
    }
}
```

我们用guard语句来检查itemTextField.text是否为空，然后把这个字符串存入text，以便在这个方法中使用。如果itemTextField.text为空，那就返回——没有需要加入待办事项列表的东西了。

接着，配置表格视图。我们创建工程所用的Xcode模版包含一条注释，告诉我们在哪里做“视图加载完成后进行额外的设置”这件事：在viewDidLoad()中。（注释中提到的nib是构建应用时故事板被编译成的格式。）添加两行代码配置表格，如代码清单27-7所示。

代码清单27-7　配置表格（ViewController.swift）

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var itemTextField: UITextField!
    @IBOutlet var tableView: UITableView!

    let todoList = TodoList()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        tableView.register(UITableViewCell.self, forCellReuseIdentifier: "Cell")
        tableView.dataSource = todoList
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func addButtonPressed(_ sender: UIButton) {
        guard let todo = itemTextField.text else {
            return
        }
        todoList.addItem(todo)
    }
}
```

我们添加的第一行告诉表格视图当数据源视图用标识"Cell"（这是我们在TodoList中所用的标识）从队列中取出一个可复用表格单元时要做什么。具体来说，就是注册了UITableViewCell这个类，让表格视图创建UITableViewCell实例。第二行告诉表格视图todoList是数据源。

还有一步就能完成控制器了。当数据源变化的时候，我们需要通知表格视图。在addButtonPressed(_:)中告诉表格视图，在我们添加一个新的待办事项到列表中之后，重新加载表格视图，如代码清单27-8所示。

代码清单27-8 通知视图重新加载数据（ViewController.swift）

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet var itemTextField: UITextField!
    @IBOutlet var tableView: UITableView!

    let todoList = TodoList()

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a nib.
        tableView.register(UITableViewCell.self, forCellReuseIdentifier: "Cell")
        tableView.dataSource = todoList
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func addButtonPressed(_ sender: UIButton) {
        guard let todo = itemTextField.text else {
            return
        }
        todoList.add(todo)
        tableView.reloadData()
    }
}
```

构建并运行应用。在文本框中输入一些文本并点击Insert按钮。现在应该能看到表格中出现待办事项了。

27.5 保存和加载 TodoList

iTahDoodle现在可以运行了。不幸的是，每次启动应用都会丢失待办事项列表。我们需要为TodoList增加保存和加载状态的能力。

27.5.1 保存 TodoList

在第26章中，我们利用NSDocument的特性实现了文档的保存和加载。一方面，基于文档的Mac应用遵循在OS X和macOS中存在多年的模式（无论是代码还是用户界面）。另一方面，大部

分iOS应用不会操作文档。iTahDoodle当然也不会——只有一个待办事项列表，而且用户期望它一直存在。

所有的iOS应用都在*应用沙盒*中运行。这意味着一个应用看不见其他应用创建的文件。沙盒的另一个副作用是，用来保存文件的目录可能会发生变化。因为要保存包含待办事项列表的文件，所以我们首先需要询问iOS在哪里保存文件。给`TodoList`添加一个从闭包计算值的属性，如代码清单27-9所示。

代码清单27-9　询问iOS在哪里保存文件（TodoList.swift）

```
import UIKit

class TodoList: NSObject {
    private let fileURL: URL = {
        let documentDirectoryURLs = FileManager.default.urls(
            for:.documentDirectory, in: .userDomainMask)
        let documentDirectoryURL = documentDirectoryURLs.first!
        return documentDirectoryURL.appendingPathComponent("todolist.items")
    }()

    fileprivate var items: [String] = []

    func add(_ item: String) {
        items.append(item)
    }
}
...
```

闭包的第一行让默认的`FileManager`（一个能让我们跟iOS文件系统交互的类）给出包含用户文档目录的URL数组。下一行从返回的数组中取出第一个元素，使用强制展开是因为iOS总是会在这个数组中返回应用的文档目录。最后，返回用户文档目录添加上`totolist.items`后的URL。

接着，添加一个方法来保存待办事项，如代码清单27-10所示。

代码清单27-10　保存待办事项列表（TodoList.swift）

```
import UIKit

class TodoList: NSObject {
    private let fileURL: URL = {
        let documentDirectoryURLs = FileManager.default.urls(
            for:.documentDirectory, in: .userDomainMask)
        let documentDirectoryURL = documentDirectoryURLs.first!
        return documentDirectoryURL.appendingPathComponent("todolist.items")
    }()

    fileprivate var items: [String] = []

    func saveItems() {
        let itemsArray = items as NSArray

        print("Saving items to \(fileURL)")
```

```
        if !itemsArray.write(to:fileURL, atomically: true) {
            print("Could not save to-do list")
        }
    }

    func add(_ item: String) {
        items.append(item)
        saveItems()
    }
}
...
```

saveItems()方法从我们刚才写的函数获取URL。接着，我们把items数组转化成NSArray。这样就能调用NSArray有而Swift数组没有的方法了。然后对NSArray调用write(to:atomically:)方法。这个方法试图把数组的内容保存到指定的URL，然后返回一个布尔型表示是否成功。

我们还在add(_:)中添加了saveItems()调用。每添加一个待办事项就保存整个列表不太理想，在即将退出应用时保存会更好；但是就iTahDoodle的目的而言，每次添加后都保存也是可以的。

构建并运行应用。在列表中添加待办事项，应该能看到每次添加完毕都有一条日志消息显示待办事项列表已保存。

27.5.2　加载 TodoList

加载已保存的待办事项列表会使用很多在保存时用到的特性。为TodoList添加一个loadItems()方法，如代码清单27-11所示。

代码清单27-11　加载已保存的待办事项列表（TodoList.swift）

```
import UIKit

class TodoList: NSObject {
    private let fileURL: URL = {
        let documentDirectoryURLs = FileManager.default.urls(
            for: .documentDirectory, in: .userDomainMask)
        let documentDirectoryURL = documentDirectoryURLs.first!
        return documentDirectoryURL.appendingPathComponent("todolist.items")
    }()

    fileprivate var items: [String] = []

    func saveItems() {
        let itemsArray = items as NSArray

        print("Saving items to \(fileURL)")
        if !itemsArray.write(to:fileURL, atomically: true) {
            print("Could not save to-do list")
        }
    }
```

```
        func loadItems() {
            if let itemsArray = NSArray(contentsOf: fileURL) as? [String] {
                items = itemsArray
            }
        }

        func add(_ item: String) {
            items.append(item)
            saveItems()
        }
    }
    ...
```

loadItems()方法首先获取我们在saveItems()中用来保存待办事项的文件URL。接着，我们尝试用初始化方法构造一个NSArray。这个初始化方法接受一个URL，表示加载数组所在的路径。如果数组可以构造，就把数组转换成[String]，然后存储在TodoList的items属性里。

现在能加载已保存的待办事项列表了，但是应该在什么时候加载呢？最简单的答案是在创建TodoList的时候尝试加载。刚才我们一直在利用没有参数的默认初始化方法，但是现在需要自己写一个显式的初始化方法了，如代码清单27-12所示。

代码清单27-12　添加一个显式的初始化方法（TodoList.swift）

```
    import UIKit

    class TodoList: NSObject {
        private let fileURL: URL = {
            let documentDirectoryURLs = FileManager.default.urls(
                for: .documentDirectory, in: .userDomainMask)
            let documentDirectoryURL = documentDirectoryURLs.first!
            return documentDirectoryURL.appendingPathComponent("todolist.items")
        }()

        fileprivate var items: [String] = []

        override init() {
            super.init()
            loadItems()
        }
        ...
    }
    ...
```

我们添加了一个新的init()，这个方法会覆盖NSObject的初始化方法。在实现中，我们调用了父类的初始化方法；必须在以任何形式访问self（举个例子，通过调用方法）之前这么做。最后，我们尝试加载已保存的待办事项。如果加载失败，TodoList会创建一个空的items数组。

完成这些后，你就成功开发出第一个iOS应用了！关于iOS开发还有很多东西要学，本章只是带你略微体验一下。

27.6 青铜挑战练习

在iTahDoodle中有几个虽然小但是很烦人的bug。首先，给列表新增待办事项时，没有清除文本框中的内容：它还留着上次的文本。其次，在文本框为空的时候点击Insert按钮可以在表格中插入空行。修复这两个bug。

27.7 白银挑战练习

`ViewController`目前承担的责任有点多。在设置表格视图时，视图控制器会注册表格单元类和复用标识，但是`TodoList`才是真正要使用新创建的表格单元的类。找到一种方法让`ViewController`在不知道`TodoList`需要用哪种表格单元的情况下设置好表格视图。

27.8 黄金挑战练习

iTahDoodle有一个非常明显的疏漏：用户无法删除待办事项！让用户通过点击待办事项将其删除。我们已经掌握了足够的知识来更新`TodoList`。关于如何检查用户何时点击了某一行，下面给出提示：让视图控制器成为表格视图的委托。这需要它符合`UITableViewDelegate`协议。`UITableViewDelegate`有一个方法有助于完成这个挑战：点击某一行就会选中那一行。

第 28 章 互 操 作

因为Swift是一门很新的语言，所以在开发中有一种很常见的情况是，需要在Objective-C实现的现有工程基础上工作。毕竟，在应用开发方面我们是有过去很多年的积累的。如果想用Swift开发原本用Objective-C开发的工程，就需要让两门语言合作。

本章要讲的就是这部分内容。如果你对Objective-C不熟悉，只要往下读，跟着敲入代码就行。不了解Objective-C不会妨碍你理解互操作的基本机制。要深入学习Objective-C，请参考最新版的《Objective-C编程》一书。

除了在现有的Objective-C工程中添加Swift代码以外，如果Swift工程需要和Cocoa或Cocoa Touch软件开发工具包（software development kit，SDK）交互，或者需要利用Objective-C，就需要用到两门语言。

正如你在第26章和第27章看到的，所有的Mac和iOS应用都会和Cocoa或Cocoa Touch框架交互。这些框架提供开发Mac和iOS应用的基本组件，而这些组件大部分是用Objective-C写成的。因此，使用这些框架就意味着和Objective-C互操作，进而影响到我们写Swift代码的方式。

要看如何互操作，我们首先用Objective-C写一个小应用。接着，在这个工程中加入一个Swift文件来定义一个Swift类并在Objective-C文件中使用。最后，创建一个Objective-C类并在Swift代码中使用。

28.1 一个 Objective-C 工程

为了说明互操作，我们要创建一个简化版的iOS联系人应用。这个工程从Objective-C开始，稍后会引入一些Swift代码。

新建Xcode工程，在模版选择窗口选中iOS，然后选择Application区域的Single View模版（如图28-1所示）。

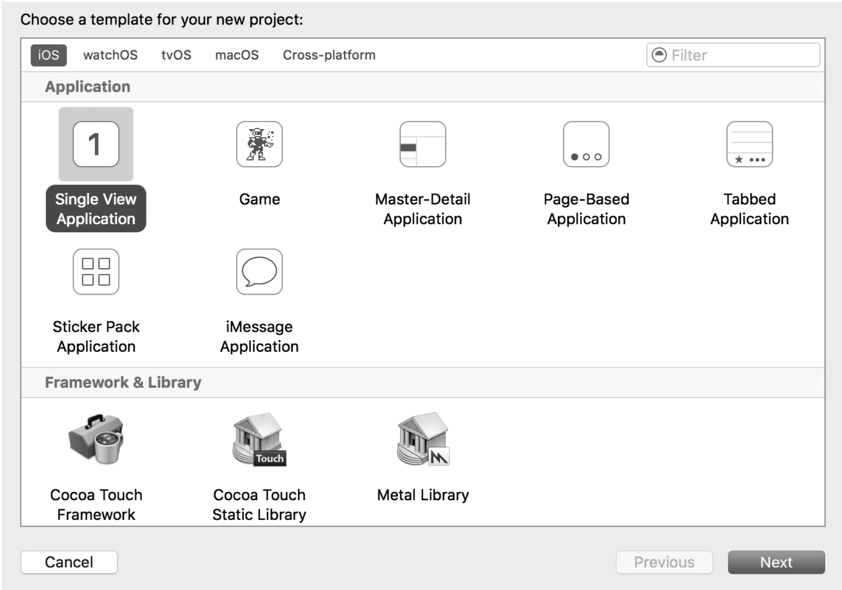


图28-1　单视图应用模版

把工程命名为Contacts，确保语言选择为Objective-C（如图28-2所示）。

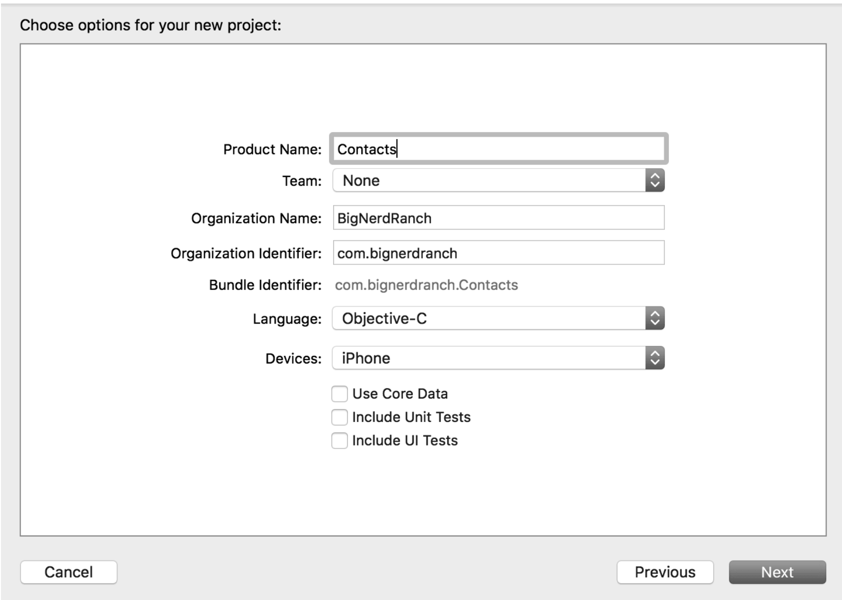


图28-2　设置工程选项

点击Next，选择一个位置保存工程，然后点击Create。

创建联系人应用

现在开始写简化版的联系人应用。先在故事板中为主视图添加一个表格，这个表格会持有一个硬编码的联系人列表。记住，故事板是Xcode提供的接口，用来管理视图以及视图之间的关系。

打开Main.storyboard。第一步是删除模版自带的视图控制器，因为我们用不到它。选中文档大纲中的视图控制器（如图28-3所示）。

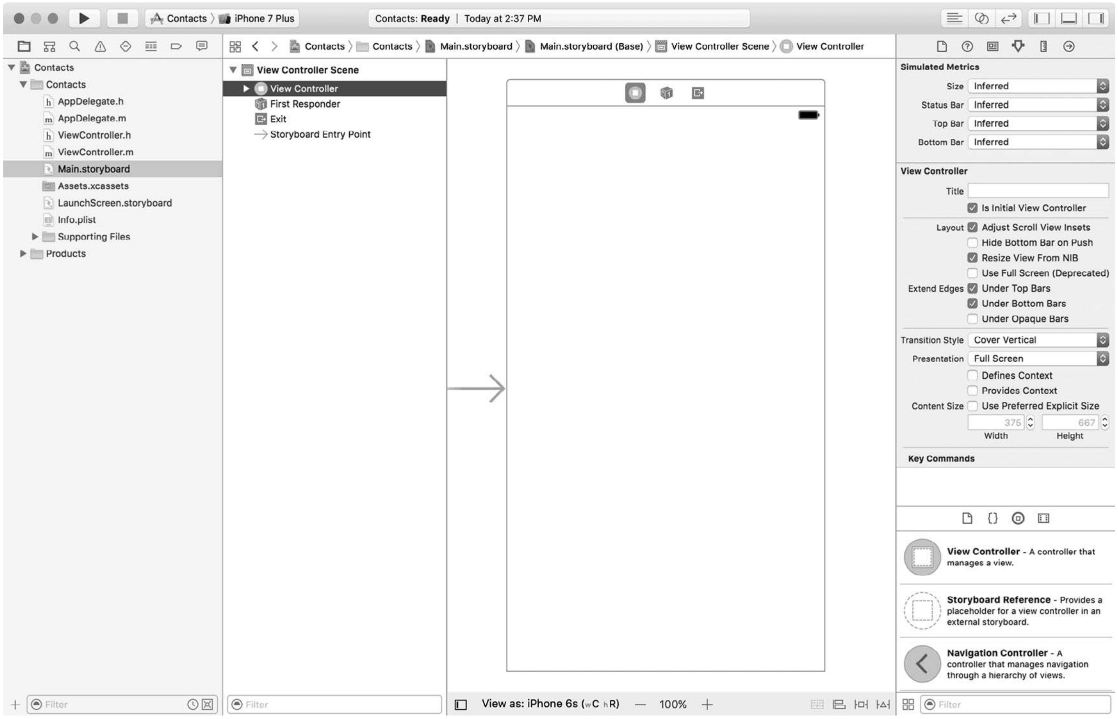


图28-3　删除视图控制器布景

选中视图控制器后，按下Delete键。

删除这个布景是必要的步骤，但是会引入一个问题。因为我们删除的布景是应用启动后的入口点，所以应用不会显示初始视图了。

事实上，如果现在试图构建并运行应用，你会发现控制台有如下信息：`Failed to instantiate the default view controller for UIMainStoryboardFile 'Main' - perhaps the designated entry point is not set?`。添加一个新的默认视图控制器可以解决这个问题。

打开对象库，搜索table，然后拖动Table View Controller到故事板画布上。

现在我们新添加了一个视图控制器到故事板，但是还没有把它设置为默认视图控制器。选择表格视图控制器，打开属性检查面板。选中会让表格视图控制器变成起始视图控制器的复选框（如图28-4所示）。

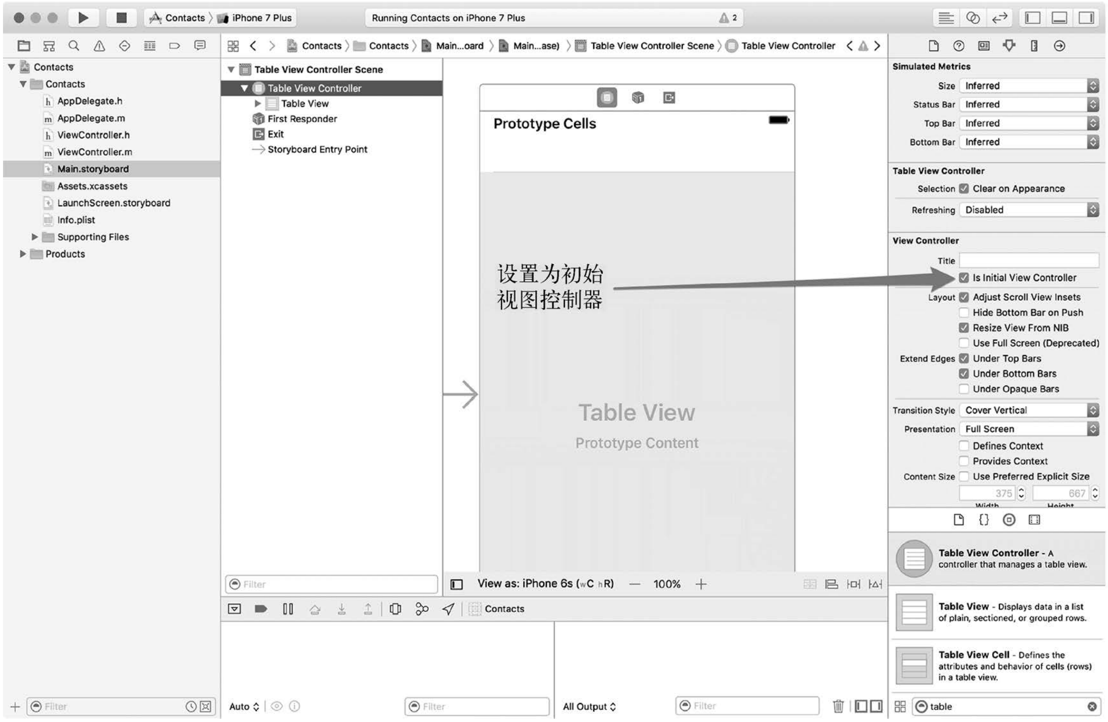


图28-4 属性检查面板

现在试着运行应用：没有错误了，只有一个空表格视图。要在这个表格视图中显示数据，需要给表格视图设置数据源，跟第27章中的方法一样。

第一步是把表格视图控制器布景和模版提供的`ViewController`关联起来。这个类原来与刚才删掉的布景相关联。我们删除那个视图控制器布景的时候断开了两者的关系，所以现在要在故事板中关联这个类和`UITableViewController`。（也可以创建一个全新的视图控制器类，但是利用模版提供的更简单。）

要利用这个类，首先把它的名字从通用的`ViewController`改为`ContactsViewController`。这个名字更具体地描述了类的目的。点击工程导航器中的ViewController.h，其内容看起来应该是这样的：

```
#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@end
```

要重命名类，在源文件中右键点击`ViewController`，在弹出菜单中选择Refactor → Rename...，把`ViewController`改成`ContactsViewController`，如图28-5所示。

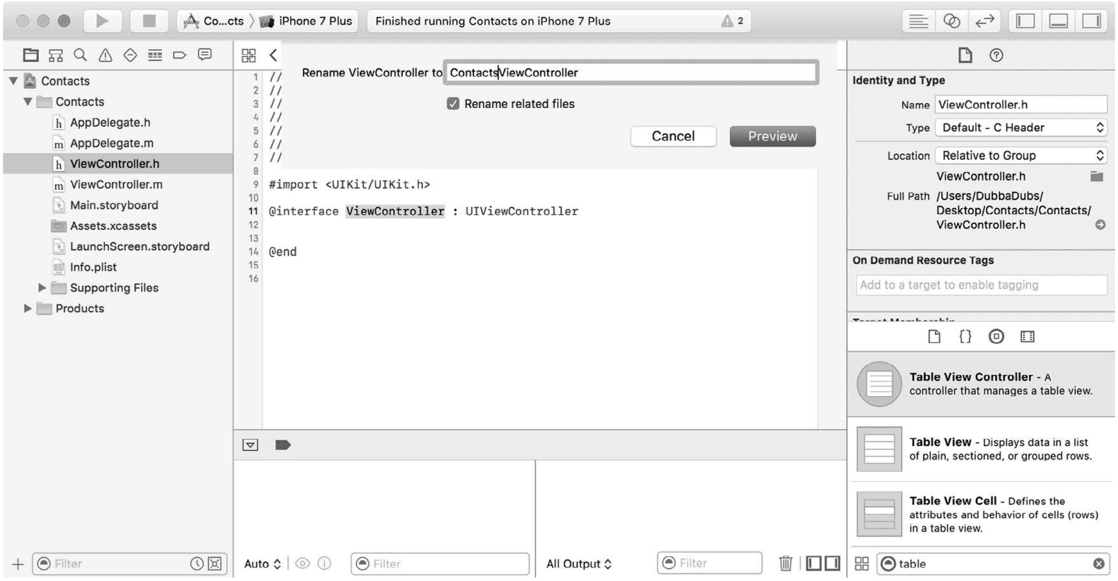


图28-5　重命名ViewController

点击Preview，这个窗口会扩大并显示即将发生的变化。点击Save。（如果Xcode提示是否打开快照，点击Enable。）

现在，我们的类有了含义更明确的名字。下一步是把它的父类改成`UITableViewController`，就是我们添加到故事板的控制器的种类，如代码清单28-1所示。

代码清单28-1　修改父类（ContactsViewController.h）

```
#import <UIKit/UIKit.h>

@interface ContactsViewController : UIViewController UITableViewController

@end
```

视图控制器的类改好之后就可以用了。它负责用联系人信息填充应用的表格视图。`ContactsViewController`继承自`UITableViewController`，所以很适合展示列表。在本例中，我们要展示的是联系人名字的列表。

要让联系人的名字在表格视图中显示，表格视图控制器需要知道应该显示哪些联系人，而且需要充当表格视图的数据源。

`UITableViewController`符合`UITableViewDataSource`协议。符合这个协议需要实现两个方法，但是在实现之前，先创建一个包含一些硬编码联系人姓名的数组。切换到ContactsViewController.m，在类扩展中为这个数组添加一个属性。在`initWithCoder:`中给数组设置一些数据，如代码清单28-2所示。

代码清单28-2 硬编码联系人姓名（ContactsViewController.m）

```
#import "ContactsViewController.h"

@interface ContactsViewController ()

@property (nonatomic, readonly, strong) NSMutableArray *contacts;

@end

@implementation ContactsViewController

- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        NSArray *contactArray = @[@"Johnny Appleseed",
                                  @"Paul Bunyan",
                                  @"Calamity Jane"];
        _contacts = [NSMutableArray arrayWithArray:contactArray];
    }
    return self;
}

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}

@end
```

可变数组`contacts`包含表示联系人的一组字符串，我们会在表格视图中显示它们。为了让这些名字出现在表格中，我们需要实现`UITableViewDataSource`协议所需的方法。添加如代码清单28-3所示的方法实现，把数据填入表格。

代码清单28-3 实现符合协议所需的方法（ContactsViewController.m）

```
...
- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    [self.tableView registerClass:[UITableViewCell class]
           forCellReuseIdentifier:@"UITableViewCell"];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}
```

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return self.contacts.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:@"UITableViewCell"
                                                  forIndexPath:indexPath];

    NSString *contact = self.contacts[indexPath.row];

    cell.textLabel.text = contact;

    return cell;
}

@end
```

这里，我们给表格视图控制器的tableView属性注册了UITableViewCell类，这样表格视图就能知道如何新建表格单元以及复用已有的表格单元了。接着，用contacts属性判断需要在tableView:numberOfRowsInSection:中提供的表格视图有多少行。我们还在tableView:cellForRowAtIndexPath:中让表格视图在必要的情况下返回一个UITableViewCell实例。在这个数据源方法中，我们用当前的indexPath把正确的联系人姓名放进表格单元实例。最后，返回要在表格视图中显示的表格单元。

构建并运行应用，联系人姓名并没有显示在表格中。我们让ContactsViewController成为了表格视图的数据源，并给了它联系人数据。问题究竟在哪里？我们少做了一件事：还没有关联故事板中的表格视图控制器布景和包含数据源代码的类。

我们可以在Main.storyboard文件中完成关联。打开故事板，点击表格视图控制器布景。确保工具区域处于显示状态。在检查面板中， 点击左起第三个图标，即标识检查面板。

找到标记着Custom Class的区域。这个区域提供一个Class字段，表示应该跟这个布景关联的自定义类。目前，这个类默认是UITableViewController，这也是我们的表格视图没有显示联系人姓名的原因。

把这个字段的值改为ContactsViewController。现在故事板看起来应该类似于图28-6。

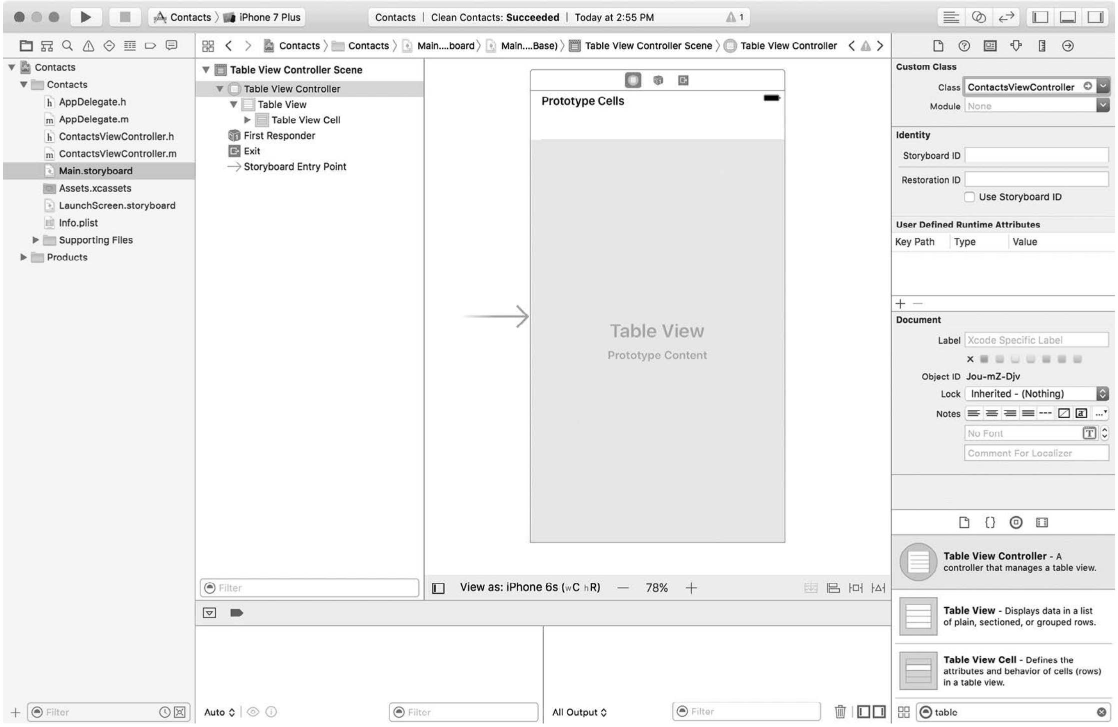


图28-6　表格视图控制器布景的自定义类

到了这里，就可以解决故事板的警告了。在工程导航器的顶部选择左起第四个图标，即问题导航器（如图28-7所示）。这个导航器会显示工程中目前存在的警告和错误。

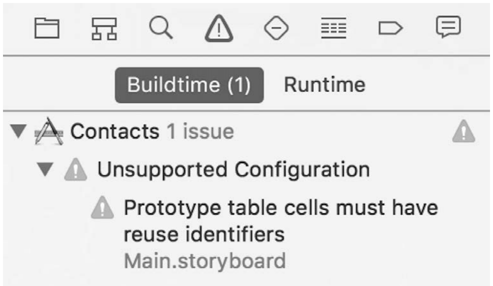


图28-7　在问题导航器中显示问题

问题导航器显示Main.storyboard有一个警告：需要给表格单元原型设置标识。因为我们不会在故事板中使用表格单元原型的特性，所以只要告诉表格视图不需要用表格单元原型就可以解决这个问题。确保文档大纲处于显示状态，展开Contacts View Controller就能看到它的层级结构了，如图28-8所示。

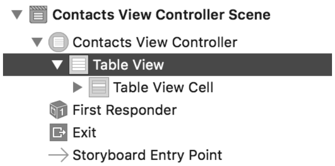


图28-8　故事板中的表格视图大纲

选择表格视图，确保工具区域处于显示状态。在检查面板中选择属性检查面板。找到Prototype Cells字段，把数字改成0（如图28-9所示）。

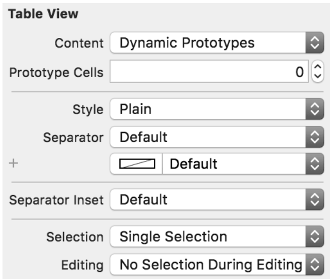


图28-9　改变故事板中的表格单元原型

警告应该消失了，构建并运行应用，应该能看到数据填入了表格视图，如图28-10所示。

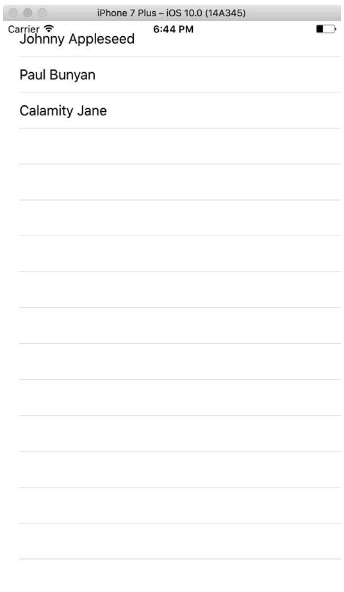


图28-10　显示联系人

28.2 在 Objective-C 工程中加入 Swift

有了一个可以运行的Objective-C简单工程，是时候加入一些Swift文件了。举个例子，现在你顿悟了：最好不要用字符串把联系人硬编码到数组中，而是应该创建一个`Contact`类型，然后在表格中显示联系人姓名。你决定用Swift写这个新类型， 因为听说它的初始化过程非常安全，想尝试一下。

在工程中添加一个新的Swift文件。点击File → New → File...。确保顶部的iOS处于选中状态，选择Source标签下的Swift File并点击Next（如图28-11所示）。

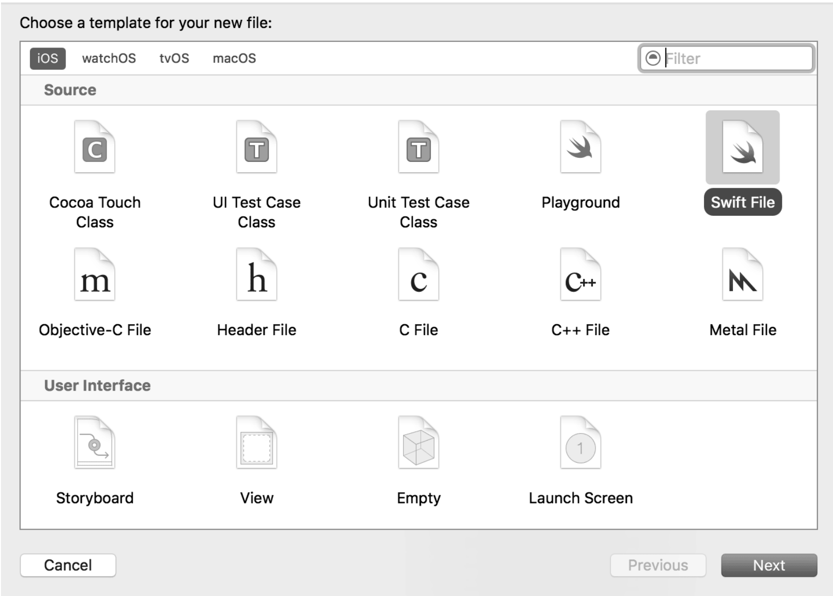


图28-11 添加新的Swift文件

在下一个窗口中，把文件命名为Contacts.swift。还要确保选中底部的复选框，把它加入Contacts目标。点击Create。

因为我们现在是在Objective-C工程中添加Swift文件，所以Xcode询问是否要配置Objective-C桥接头文件（bridging header，如图28-12所示）。

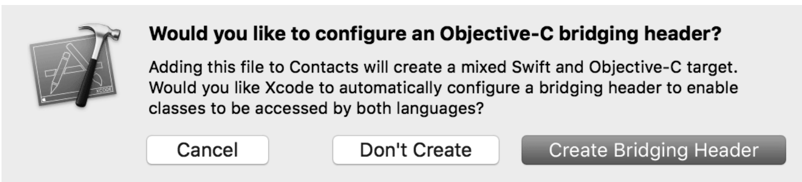


图28-12 要配置Objective-C桥接头文件吗

28

桥接头文件用来从Objective-C代码“桥接”到Swift代码。点击Create Bridging Header，Xcode会创建新的Swift文件Contact.swift以及Contacts-Bridging-Header.h——即Objective-C桥接头文件。

在工程导航器中找到并选择Contacts-Bridging-Header.h。你会看到里面除了几行注释以外没有别的内容，如代码清单28-4所示。

代码清单28-4　查看Contacts-Bridging-Header.h

```
//
//  Use this file to import your target's public headers
//  that you would like to expose to Swift.
//
```

如你所见，这个文件里什么都没有。稍后会往里添加一句代码用来引入一个Objective-C类的头文件，而我们想把这个类暴露给Swift。这就是桥接头文件的作用。

不过那是后面的事情。现在，我们要创建`Contact`类。切换到Contact.swift，添加如代码清单28-5所示的代码。

代码清单28-5　创建`Contact`类（Contact.swift）

```
import Foundation

class Contact: NSObject {
    let name: String
    init(name: String) {
        self.name = name
        super.init()
    }
}
```

这个新的类继承自`NSObject`，这样就可以把它暴露给应用的Objective-C部分了。要从Objective-C调用Swift，继承Objective-C类这一步是必要的。在本例中，我们从`NSObject`继承，它是Objective-C中的基类。`Contact`的实现很简单，只有一个`name`属性和一个初始化方法，而且这个初始化方法只有一个`name`参数用来配置实例。

如果只用Swift写这个工程的话，用结构体实现`Contact`会更好。不过，这对于Objective-C/Swift混编的工程行不通，因为Swift的结构体对Objective-C不可见。

现在写好了类，就可以在Objective-C代码中使用了。我们修改`ContactsViewController`，让它维护一个`Contact`对象的数组，而不是字符串数组。

首先，在ContactsViewController.m中引入刚才Xcode生成的头文件，如代码清单28-6所示。这样就可以在表格视图控制器中使用Swift版的`Contact`类了。

代码清单28-6　引入Contacts-Swift.h头文件（ContactsViewController.m）

```
#import "ContactsViewController.h"
#import "Contacts-Swift.h"

@interface ContactsViewController ()
```

```
@property (nonatomic, readonly, strong) NSMutableArray *contacts;

@end
...
```

头文件Contacts-Swift.h向引用它的文件暴露刚才写的Swift代码。它包含的接口能把Swift代码共享给应用的Objective-C组件。

这类头文件的命名规范是：ProductModuleName-Swift.h。这里，产品名是Contacts；因为这个应用是单目标应用，所以模块名也是它。因此生成的头文件是Contacts-Swift.h。

既然`Contact`类在ContactsViewController.m中可见了，那就可以把`contacts`数组改为保存`Contact`实例而不是硬编码的字符串。还有，确保在数据源方法`tableView:cellForRowAtIndexPath:`中使用联系人的`name`，如代码清单28-7所示。

代码清单28-7 更新联系人数组（ContactsViewController.m）

```
#import "ContactsViewController.h"
#import "Contacts-Swift.h"

@interface ContactsViewController ()

@property (nonatomic, readonly, strong) NSMutableArray *contacts;

@end

@implementation ContactsViewController

- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        NSArray *contactArray = @[@"Johnny Appleseed",
                                  @"Paul Bunyan",
                                  @"Calamity Jane"];
        Contact *c1 = [[Contact alloc] initWithName: @"Johnny Appleseed"];
        Contact *c2 = [[Contact alloc] initWithName: @"Paul Bunyan"];
        Contact *c3 = [[Contact alloc] initWithName: @"Calamity Jane"];
        _contacts = [NSMutableArray arrayWithArray:contactArray @[c1, c2, c3]];
    }
    return self;
}
...

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:@"UITableViewCell"
                                                  forIndexPath:indexPath];

    NSStringContact *contact = self.contacts[indexPath.row];

    cell.textLabel.text = contact.name;

    return cell;
}
```

构建并运行应用，在模拟器中看到的结果应该跟之前一样。

添加联系人

我们已经在使用新的Swift类型了，但是还在硬编码联系人姓名。这种做法显然不会长久——如果用户要添加联系人怎么办？我们的应用需要一种能新建联系人的机制。

首先删除硬编码的联系人。取而代之的是初始化一个空的contacts数组，稍后用数据填充，如代码清单28-8所示。

代码清单28-8　把硬编码的联系人替换为数组（ContactsViewController.m）

```
...
- (id)initWithCoder:(NSCoder *)aDecoder {
    self = [super initWithCoder:aDecoder];
    if (self) {
        Contact *c1 = [[Contact alloc] initWithName: @"Johnny Appleseed"];
        Contact *c2 = [[Contact alloc] initWithName: @"Paul Bunyan"];
        Contact *c3 = [[Contact alloc] initWithName: @"Calamity Jane"];
        _contacts = [NSMutableArray arrayWithArray: @[c1, c2, c3]];
       _contacts = [NSMutableArray array];
    }
    return self;
}
...
```

接着，新建一个文件，用于存放创建联系人的视图控制器。

当Xcode询问新文件的模版时，选择iOS下面的Source，然后选择Cocoa Touch Class。我们要继承UIKit中的一个类，UIKit只在iOS工程中能用。把这个新文件命名为NewContactsViewController，并使它继承UIViewController。还有，确保选择Swift语言（图28-13）。

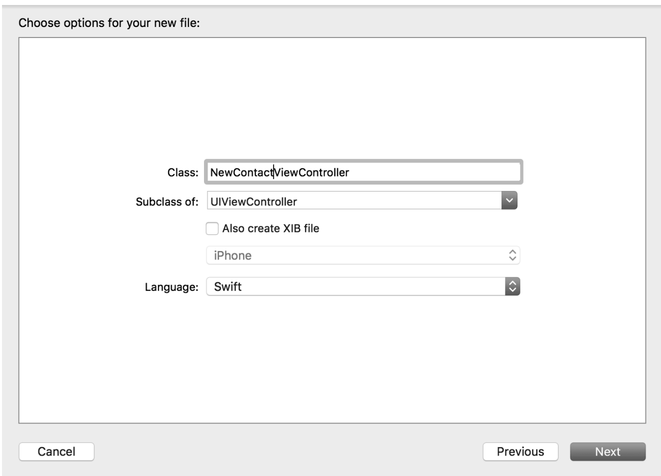


图28-13　继承UIViewController

点击Next，确保选择工程目标后再创建文件。

我们稍后再补充新类`NewContactViewController`的细节。现在先切换到Main.storyboard选择联系人视图控制器。我们需要在视图中添加控件，让用户可以新建联系人。在视图顶部添加导航栏，上面有一个按钮会启动`NewContactViewController`。

最简单的方式是添加一个`UINavigationController`作为故事板的初始视图控制器。在对象库中搜索导航控制器，拖动一个到故事板的画布上。

把`UINavigationController`放到故事板上会产生一个导航控制器，它有一个根视图控制器布景。我们并不需要这个根视图控制器布景，其实我们已经创建了一个根视图控制器——联系人视图控制器。删除根视图控制器布景。

接着，把导航控制器设置为初始视图控制器。在文档大纲中，展开Navigation Controller Scene，选择Navigation Controller。打开属性检查面板，并勾选Initial View Controller。

注意，这样修改以后应用就不能显示联系人视图控制器了。在导航视图控制器和联系人视图控制器之间创建关系连接（relationship segue）可以解决这个问题。选择导航控制器布景，按住Control键并拖动到联系人视图控制器。在弹出菜单的Relationship Segue区域中选择root view controller（如图28-14所示）。

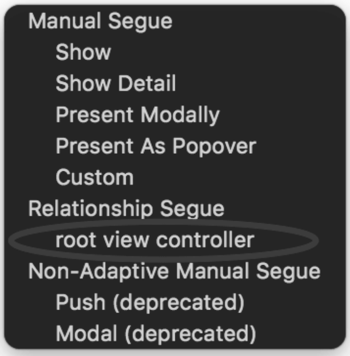


图28-14　设置根视图控制器

现在应用主界面运行在导航控制器内了。这意味着在导航控制器内的每个视图控制器都有一个可配置的`UINavigationItem`，通过它可以访问`title`属性；可以在故事板中设置`title`属性。在文档大纲中点击联系人视图控制器的导航控件。在属性检查面板中找到这个控件的title字段，输入Contacts。

如果构建并运行应用，你还是会看到空的表格视图，但它现在嵌在一个导航控制器中，并且标题是Contacts。

接着，在导航栏添加一个按钮。在对象库中搜索“UIBarButtonItem”。拖动这个类型的一个实例到联系人视图控制器布景的右导航控件处（如图28-15所示）。

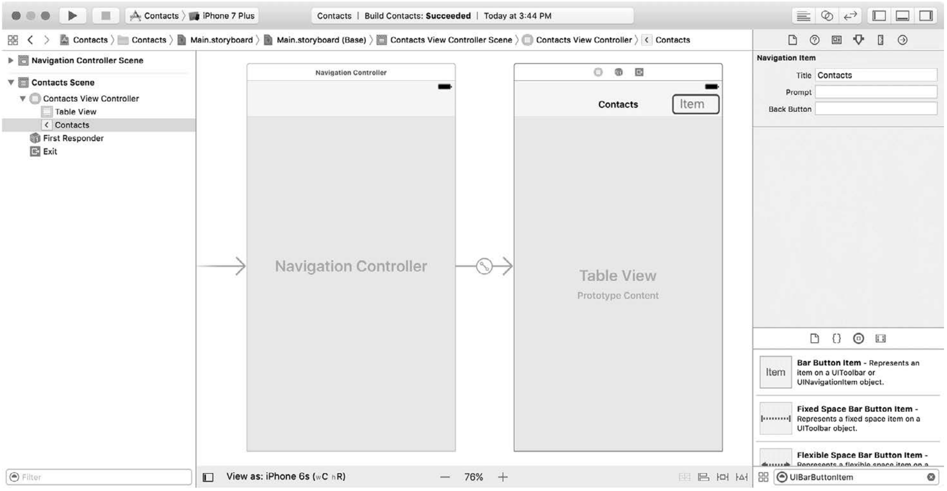


图28-15　添加一个按钮作为导航控件

选择`UIBarButtonItem`并改变其外观。在属性检查面板中，把按钮的标识从Custom改成Add。这样按钮就变成了加号图标。

我们要让这个按钮控件启动一个新建联系人的视图。从对象库中拖动一个新的视图控制器到故事板。点击视图控制器，在标识检查面板中把它的类名改为`NewContactViewController`，这样可以将这个视图控制器和我们新创建的Swift类关联起来。在文档大纲中，这个视图控制器布景会被重命名为New Contact View Controller Scene。

现在我们需要添加一些标签和文本框到新建联系人视图控制器中，这样用户才可以新建联系人。从对象库拖动两个`UILabel`实例和两个`UITextField`到故事板。如图28-16所示设置视图。

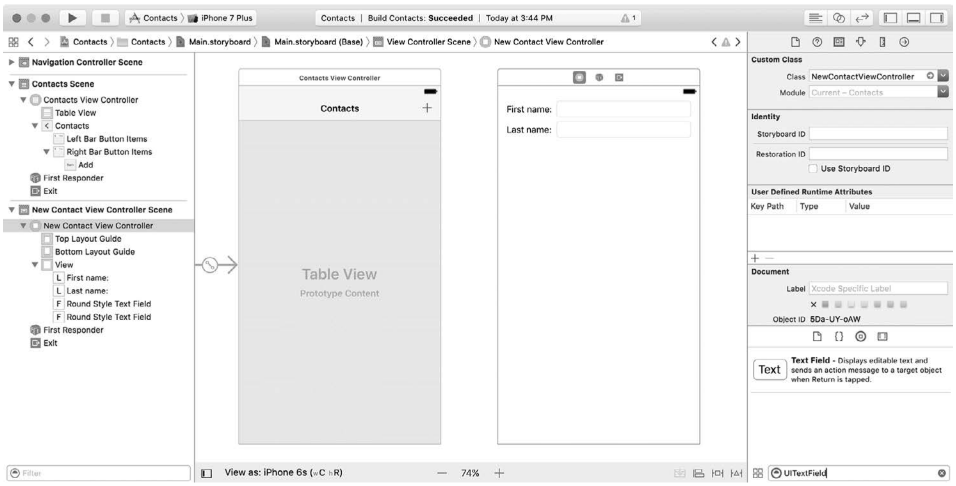


图28-16　新建联系人视图控制器

接着，我们需要连接表格视图控制器导航栏上的`UIBarButtonItem`和`NewContactViewController`。按住Control键，从加号按钮拖动到新建联系人视图控制器布景。放开鼠标，会弹出一个菜单询问要创建什么样的动作连接（Action Segue）。选择Present Modally。

构建并运行应用，点击加号按钮，你会发现应用会模态化展示`NewContactViewController`的视图。

目前为止一切正常，但是还没有结束。输入联系人信息后，用户无法关掉这个视图，也没有办法保存新建的联系人。下一步是让用户能保存新建的联系人并关掉这个视图控制器。

我们可以在视图控制器上添加一个按钮让用户保存输入的联系人信息并关闭`NewContactViewController`，但是这样一个按钮和导航栏上已有的按钮放在一起有点不协调。因此，要把`NewContactViewController`嵌入导航控制器中，然后为新建联系人视图控制器添加一个按钮作为导航控件。事实上，我们要添加两个按钮作为导航控件：一个保存新建的联系人，另一个取消这个流程。

从对象库拖动一个新的`UINavigationController`并放到Main.storyboard的画布上。跟之前一样，用已有的New Contact View Controller替换导航控制器的根视图控制器。删除已有的根视图控制器，按住Control键从导航控制器拖动到`NewContactViewController`，选择root view controller关系连接。

现在`NewContactViewController`嵌入了`UINavigationController`，就可以跟前面设置`ContactsViewController`的导航控件标题一样进行设置了。选择`NewContactViewController`的导航控件，把标题改为Contact。

虽然我们把新建联系人视图控制器作为新建的导航控制器的根视图控制器，但是如果现在运行应用，新建联系人视图控制器并不会在导航控制器中显示。联系人视图控制器并不知道新的导航控制器存在。想解决这个问题，要把`ContactsViewController`的加号按钮的连接改成连接到新的导航控制器。

按住Control键并点击`ContactsViewController`的加号按钮，点击Contact旁边的小叉号按钮删除连接，如图28-17所示。

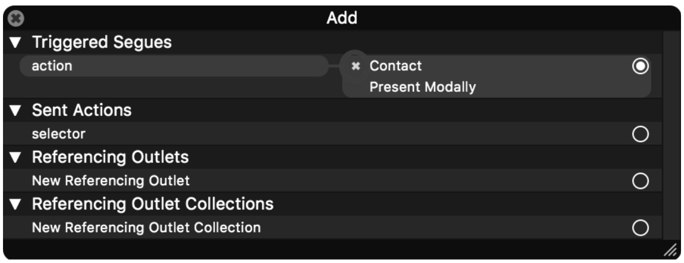


图28-17 删除连接

接着，建立一个连接来显示`UINavigationController`。按住Control键从加号按钮拖动到`UINavigationController`。选择模态化展示导航控制器的选项。

现在，我们可以给NewContactViewController的导航栏添加两个UIBarButtonItem的实例了。从对象库拖出两个实例，一个放在导航栏的左边，一个放在导航栏的右边。选择左边的按钮，在属性检查面板中把System Item改成Cancel，再用同样的方式把右边的按钮改成Save。现在故事板的布局看起来应该如图28-18所示。

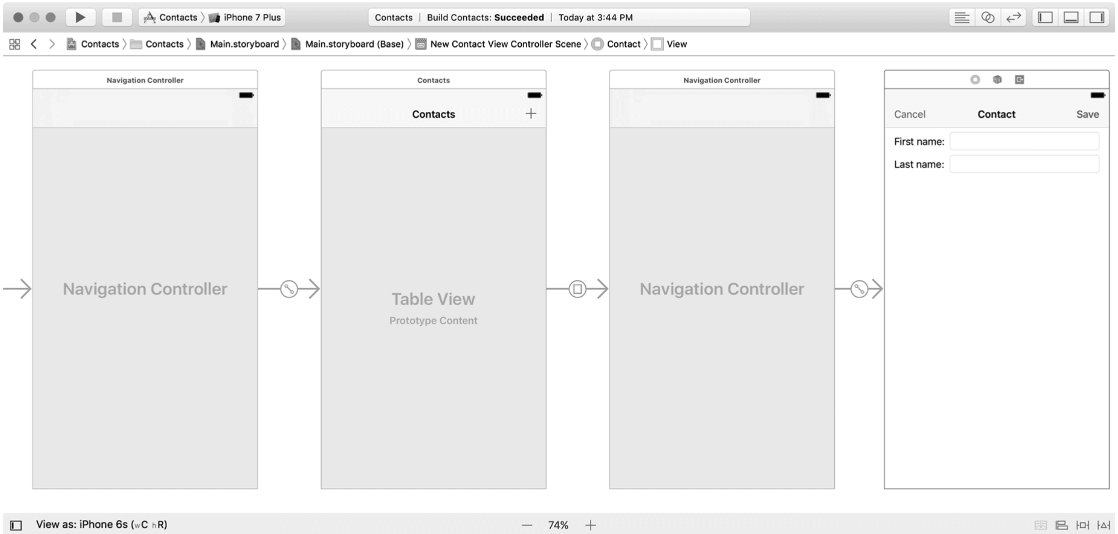


图28-18　新的故事板布局

在设置两个按钮的动作前，要先在NewContactViewController 中为名字文本框和姓氏文本框创建两个出口（outlet）。这两个出口可以让我们访问文本框的文本，当用户点击Save按钮时，我们要利用这些文本新建Contact类型的实例。打开NewContactViewController.swift，并为两个文本框添加如代码清单28-9所示的出口属性。

代码清单28-9　为文本框添加出口（NewContactViewController.swift）

```
import UIKit

class NewContactViewController: UIViewController {
    @IBOutlet var firstNameTextField: UITextField!
    @IBOutlet var lastNameTextField: UITextField!
    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

为这些文本框添加了@IBOutlet属性后，需要在故事板中连接起来。打开Main.storyboard，在新建联系人布景中连接这些出口。按住Control键从新建联系人视图控制器拖动到每个文本框，选择对应的IBOutlet，就跟在第26章中所做一样。

属性连接到对应的UITextField之后，就可以给两个按钮设置动作，让它们创建联系人或取消创建。我们会利用UIStoryboard的一个特性，叫作*回退连接*（unwind segue）。回退连接可以在导航栈中的一个视图控制器和它之前的那个视图控制器之间建立关系。你可以把回退连接理解为一种创建*回退导航*（backwards navigation）的机制。当我们想用这些按钮回到用户的联系人列表时，故事板的这个特性很方便。

要使用回退连接，必须先在作为回退目的的视图控制器上写一个回退方法。在本例中，当用户取消新建联系人操作时，我们要回退到联系人视图控制器，所以要在ContactsViewController上添加方法。

打开ContactsViewController.m并添加这个方法，如代码清单28-10所示。

代码清单28-10　为取消操作添加方法（ContactsViewController.m）

```
...
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    ...
}

- (IBAction)cancelToContactsViewController:(UIStoryboardSegue *)segue
{
    // 如果用户取消操作的话就什么都不用做
}

@end
```

新方法cancelToContactsViewController:接受UIStoryboardSegue的一个实例作为参数。还要注意，我们通过IBAction把这个方法暴露给了故事板。segue参数附带很多有用的信息，我们会利用这些信息找到新联系人的姓名。

我们需要在Main.storyboard中回退连接到新建联系人视图控制器。打开故事板，选择这个视图控制器。看到联系人布景顶部有个Exit图标（如图28-19所示）了吗？我们会利用这个元素连接到回退动作，从而关闭NewContactViewController。

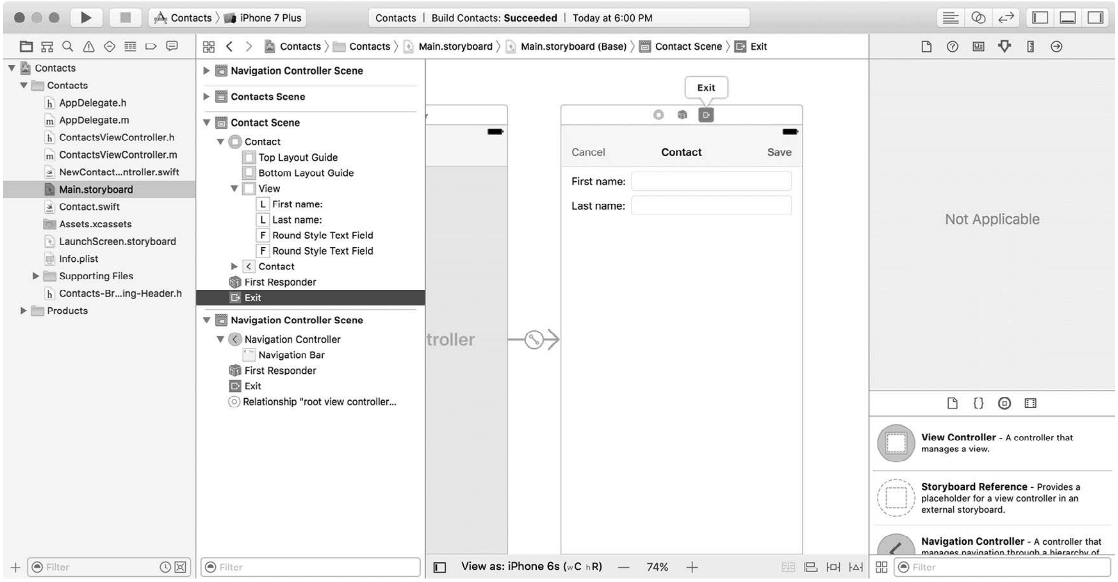


图28-19　从布景退出

按住Control键从Cancel按钮控件拖动到Exit图标。放开鼠标，你会看到一个选项可以把按钮的动作连接到`cancelToContactsViewController:`。选择这个方法。

运行应用，点击按钮新建联系人，然后点击Cancel。这样就会回退到联系人列表了。

下一步是把Save按钮连接到保存新建联系人信息的动作，并回退到联系人列表。此外，还需要更新联系人列表让它显示新建的联系人。我们采取的步骤跟对待Cancel按钮一样，不过这次会创建一个回退动作。这个回退动作会利用`UIStoryboardSegue`实例携带的信息。

打开ContactsViewController.m，添加一个回退动作，如代码清单28-11所示。

代码清单28-11　添加`createNewContact:`（ContactsViewController.m）

```
...

- (IBAction)cancelToContactsViewController:(UIStoryboardSegue *)segue
{
    // 如果用户取消操作的话就什么都不用做
}

- (IBAction)createNewContact:(UIStoryboardSegue *)segue
{
    NewContactViewController *newContactVC = segue.sourceViewController;
    NSString *firstName = newContactVC.firstNameTextField.text;
    NSString *lastName = newContactVC.lastNameTextField.text;
    if (firstName.length != 0 || lastName.length != 0) {
        NSString *name = [NSString stringWithFormat:@"%@ %@",
                                   firstName, lastName];
        Contact *newContact = [[Contact alloc] initWithName:name];
```

```
            [self.contacts addObject:newContact];
            [self.tableView reloadData];
        }
    }
    @end
```

这里我们定义了一个回退操作。这个操作利用segue参数得到发起回退的sourceView-Controller，然后就可以从NewContactViewController的UITextField属性出口获取文本了。

接着，确保至少有一个字符串非空。如果有文本，就可以新建一个Contact的实例，把它加入contacts属性。最后，重新加载tableView以显示新建联系人的姓名。

现在，当用户点击Save按钮来新建联系人时，就可以把这个方法用作回退操作了。回到Main.storyboard，选择新建联系人视图控制器。按住Control键从Save按钮拖动到Exit图标，选择createNewContact:。

运行应用，新建一个联系人并点击Save按钮。这样应该会回到联系人列表，在这里能看到刚才新建的联系人。

28.3 添加 Objective-C 类

我们已经实践了从Objective-C到Swift的互操作，下一个任务是实践从Swift到Objective-C的互操作。我们要新建一个Objective-C类为新建的联系人生成默认头像。Swift类NewContactView-Controller会用到这个新的Objective-C类。这模拟了一种常见的场景：已有的工程总是会有一些Objective-C类需要在其Swift部分使用。

新建一个Cocoa Touch Class类型的Objective-C文件，命名为ImageFactory。它的作用是为新建的联系人创建默认头像。让这个类继承NSObject。确保选择Objective-C作为这个类的语言。

在为这个类写代码之前，需要先在Main.storyboard中为联系人布景器添加一个UIImageView。这个图像视图会显示新建联系人的默认头像。从对象库拖动一个Image View到联系人布景上。把图像视图的尺寸设置为240点×240点。

接着添加自动布局约束，确保NewContactViewController会正确显示子视图。

首先利用垂直和水平约束把图像视图放在中央。选择UIImageView，打开故事板右下角的自动布局Align菜单。在菜单中选中Horizontally in Container和Vertically in Container，如图28-20所示。点击Add 2 Constraints按钮。

接着，为图像视图设置宽度和高度约束。保持图像视图选中的状态，选择故事板右下角的自动布局Pin菜单。选中Width和Height，保持其值不变（如图28-21所示）。添加这两个约束，固定图像视图的宽度和高度。

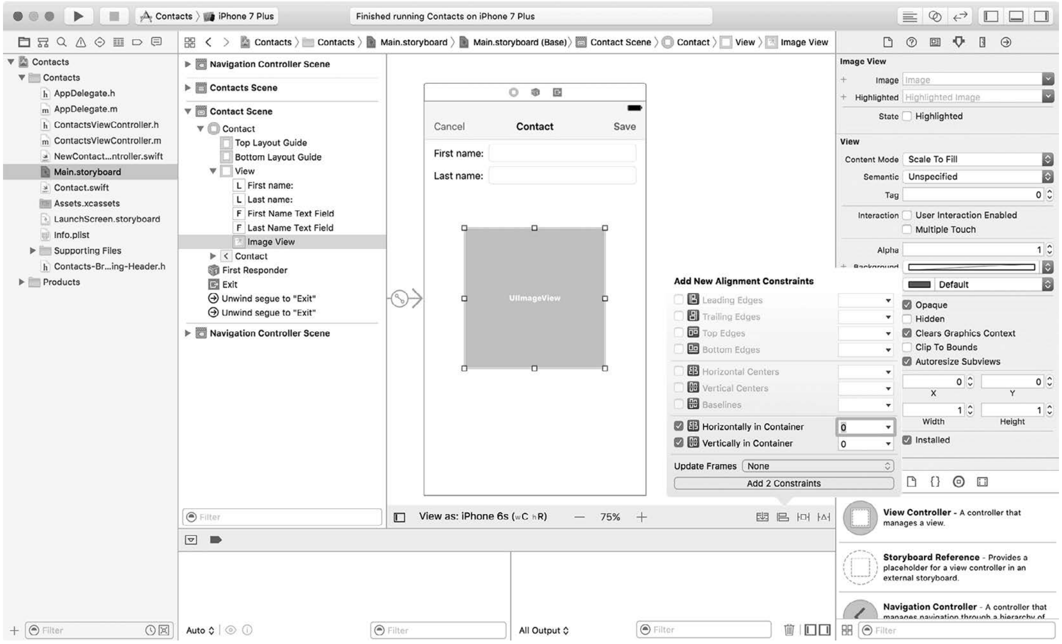


图28-20　让图像视图居中

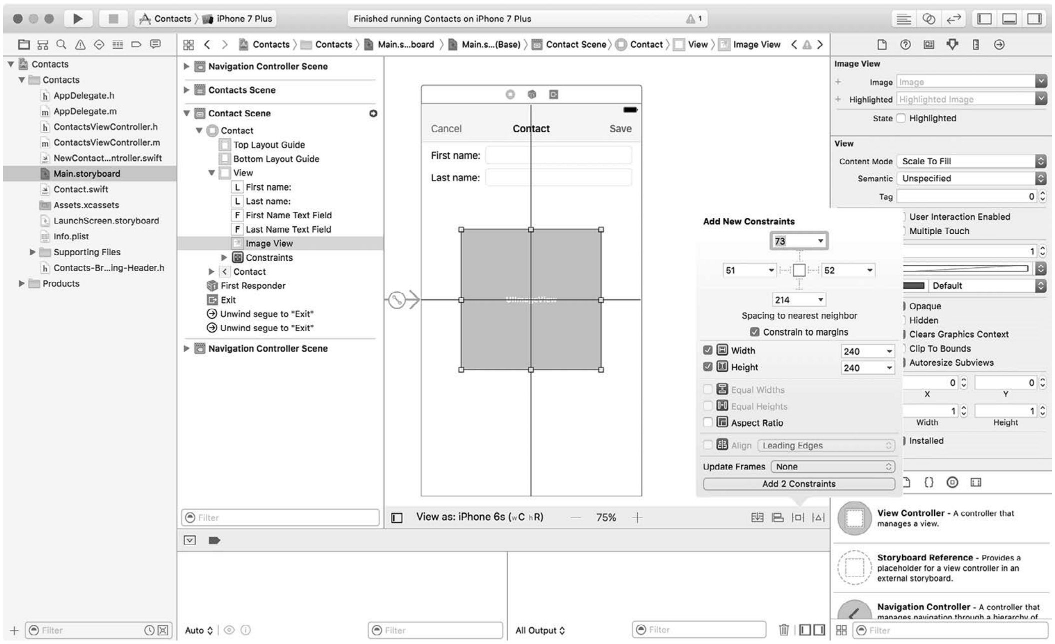


图28-21　图像视图的宽度和高度约束

完成之后，联系人布景看起来应该类似于图28-22。

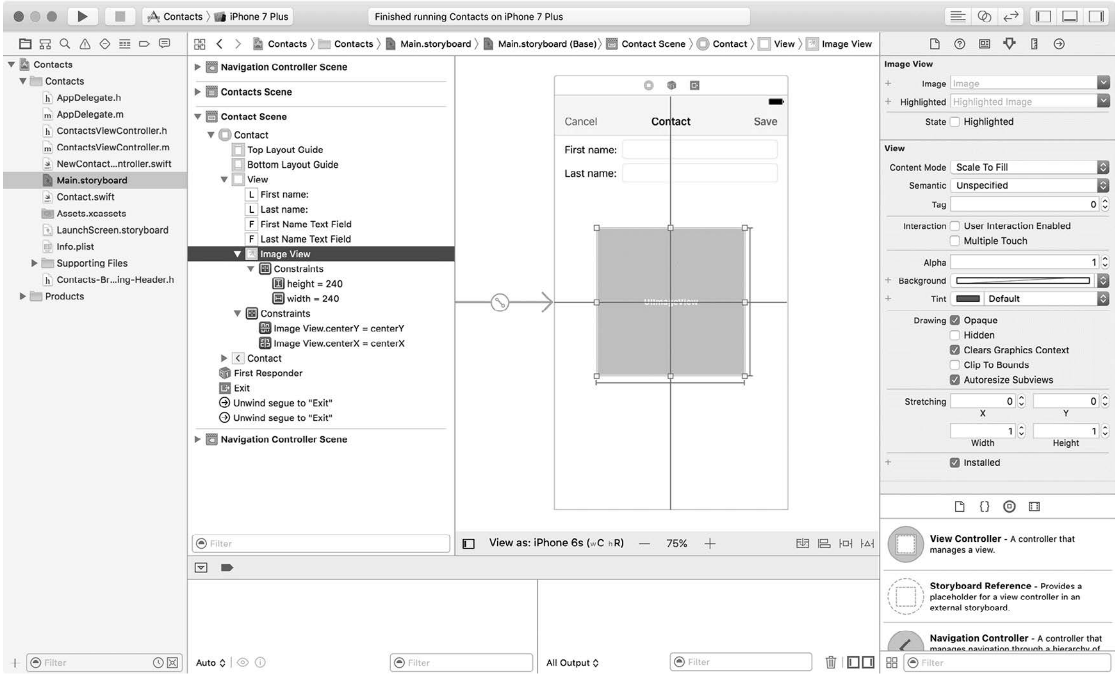


图28-22　新建联系人视图的自动布局

这些简单的约束会让视图在iPhone 7和iPhone 7 Plus模拟器中的竖屏模式下正常显示。如果改变设备或朝向，布局就会变化。

现在联系人布景上有了图像视图，还需要在`NewContactViewController`中为图像视图添加`IBOutlet`，如代码清单28-12所示。在`NewContactViewController`中添加`UIImageView`属性，这样就可以设置要显示的图片了。

代码清单28-12　添加一个连接到图像视图的`IBOutlet`（NewContactViewController.swift）

```
...
class NewContactViewController: UIViewController {
    @IBOutlet var firstNameTextField: UITextField!
    @IBOutlet var lastNameTextField: UITextField!
    @IBOutlet var contactImageView: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.

    }
```

```
    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

最后，确保这个属性连接到了Main.storyboard中的图像视图。

现在该实现ImageFactory类了。切换到ImageFactory.h，添加一个generateDefaultImageOfSize:方法，如代码清单28-13所示。确保在文件开头引入了UIKit。

代码清单28-13　实现ImageFactory（ImageFactory.h）

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@interface ImageFactory : NSObject

+ (UIImage *)generateDefaultImageOfSize:(CGSize)size;

@end
```

这个新方法是ImageFactory类的公开接口，它会利用size参数进行一些屏幕外的绘制。打开ImageFactory.m，敲入绘制代码，如代码清单28-14所示。这块代码很长，慢慢来。

代码清单28-14　绘制默认头像（ImageFactory.m）

```
#import "ImageFactory.h"

@implementation ImageFactory

+ (UIImage *)generateDefaultImageOfSize:(CGSize)size
{
    // 创建frame并获取context
    CGRect frame = CGRectMake(0, 0, size.width, size.height);
    UIGraphicsBeginImageContext(size);
    CGContextRef context = UIGraphicsGetCurrentContext();

    // 绘制白色背景和黄色圆圈
    CGContextSetFillColorWithColor(context, [[UIColor whiteColor] CGColor]);
    CGContextFillRect(context, frame);
    CGContextSetFillColorWithColor(context, [[UIColor yellowColor] CGColor]);
    CGContextFillEllipseInRect(context, frame);

    CGFloat x = frame.origin.x + size.width / 2;
    CGFloat y = frame.origin.y + size.height / 2;
    CGPoint center = CGPointMake(x, y);

    // 绘制眼睛和微笑
    CGSize eyeSize = CGSizeMake(frame.size.width * 0.1, frame.size.height * 0.1);
    CGFloat separation = frame.size.width * 0.1;

    CGPoint leftPt = CGPointMake(center.x - (separation + eyeSize.width),
        center.y - (eyeSize.height * 2));
```

```
    CGPoint rightPt = CGPointMake(center.x + separation,
        center.y - (eyeSize.height * 2));

    CGRect leftEye = CGRectMake(leftPt.x, leftPt.y, eyeSize.width, eyeSize.height);
    CGRect rightEye = CGRectMake(rightPt.x, rightPt.y, eyeSize.width, eyeSize.height);
    CGContextSetFillColorWithColor(context, [[UIColor blackColor] CGColor]);
    CGContextFillEllipseInRect(context, leftEye);
    CGContextFillEllipseInRect(context, rightEye);

    CGFloat smileHeight = center.y + eyeSize.height;
    CGPoint leftEdge = CGPointMake(leftEye.origin.x, smileHeight);
    CGPoint rightEdge = CGPointMake(rightEye.origin.x + rightEye.size.width, smileHeight);
    CGFloat smileLength = rightEdge.x - leftEdge.x;
    CGFloat smileWidth = eyeSize.width / 3;

    CGContextSetLineWidth(context, smileWidth);
    CGContextBeginPath(context);
    CGContextMoveToPoint(context, leftEdge.x, leftEdge.y);
    CGContextAddCurveToPoint(context, leftEdge.x + (smileLength / 4),
        smileHeight + smileWidth * 2, rightEdge.x - (smileLength / 4),
        smileHeight + smileWidth * 2, rightEdge.x, rightEdge.y);
    CGContextStrokePath(context);

    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return image;
}
@end
```

绘制代码可能很难看懂，不过不用太担心。这里用到的框架叫Core Graphics，其细节超出了本书的讨论范围。如果感到好奇，苹果关于Core Graphics的文档对每个函数的作用都有很详尽的介绍。最后的结果是一张黄色的笑脸，我们会把它用作联系人的默认头像。

`generateDefaultImageOfSize:`的主要作用是创建一个`UIImage`的实例。要做到这一点，首先拿到一个大小合适的图形上下文，接着获取这个上下文的引用。这样就可以利用一系列的Core Graphics绘制函数了。我们用`UIGraphicsGetImageFromCurrentImageContext()`从当前上下文创建一幅图像。因为新建了一个上下文，所以还得清理绘制环境。最后，返回创建的图像。

在Swift中与Objective-C互操作时，可以思考一下工程中的这两门语言是怎么互相通信的。举个例子，`generateDefaultImageOfSize:`方法为Swift代码提供了一个简单的接口来生成默认头像。

在Swift中使用Objective-C类之前，需要先在桥接头文件中引入这个类。打开Contacts-Bridging-Header.h并引入`ImageFactory`的头文件，如代码清单28-15所示。这样做能让Objective-C类在Swift代码中可用。

28

代码清单28-15 在桥接头文件中引入Objective-C类的头文件（Contacts-Bridging-Header.h）

```
//
// Use this file to import your target's public headers
// that you would like to expose to Swift.
```

```
//
#import "ImageFactory.h"
```

一旦ImageFactory对Swift可见，就可以使用了。在NewContactViewController的viewDidLoad方法中新建一个ImageFactory的实例，如代码清单28-16所示。

代码清单28-16　在NewContactViewController中使用Objective-C的ImageFactory类（NewContactViewController.swift）

```
...
class NewContactViewController: UIViewController {
    @IBOutlet var firstNameTextField: UITextField!
    @IBOutlet var lastNameTextField: UITextField!
    @IBOutlet var contactImageView: UIImageView!
    override func viewDidLoad() {
        super.viewDidLoad()

        // Do any additional setup after loading the view.
        contactImageView.image =
            ImageFactory.generateDefaultImage(of: contactImageView.frame.size)
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

在Contacts-Bridging-Header.h中添加ImageFactory.h意味着这个文件对同一个目标中的任何Swift文件都是可见的。使用ImageFactory类时可以把它当作采用Swift所写的，就像上面对viewDidLoad()的实现中那样。

除了在桥接头文件中引入ImageFactory.h以外，不需要做任何其他事情让ImageFactory在Swift代码中可用，直接调用ImageFactory的方法就行，就好像它是个Swift类。这样做会触发ImageFactory中的绘制代码并返回图像。返回的图像被赋给了contactImageView的image属性。

Swift把Objective-C方法generateDefaultImageOfSize:翻译为generateDefaultImage(of:)。Swift语法比Objective-C更加在意命名的简洁性，所以这个方法在Swift中去掉了size，并且把of放在括号中。单词size或者其他类似的单词一般会用作传递给这个方法的值的名字，如代码清单28-16所示。Swift风格建议从方法名中去掉可能和方法参数产生冗余的名字。

按住Option键并点击NewContactViewController.swift中的generateDefaultImage(of:)可以看到方法是如何暴露给Swift的（如图28-23所示）。

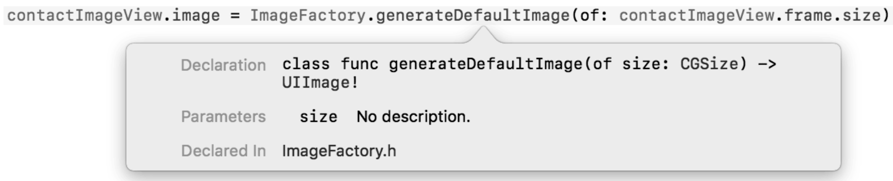


图28-23 Swift中的`generateDefaultImageOfSize:`

注意Swift把Objective-C的参数名`size`翻译成了内部参数：`generateDefaultImage(of size: CGSize)`。

运行应用就可以看到我们的成果了。点击右上角的加号按钮新建联系人。`NewContactViewController`会显示出来，文本框下面就是默认头像（如图28-24所示）。

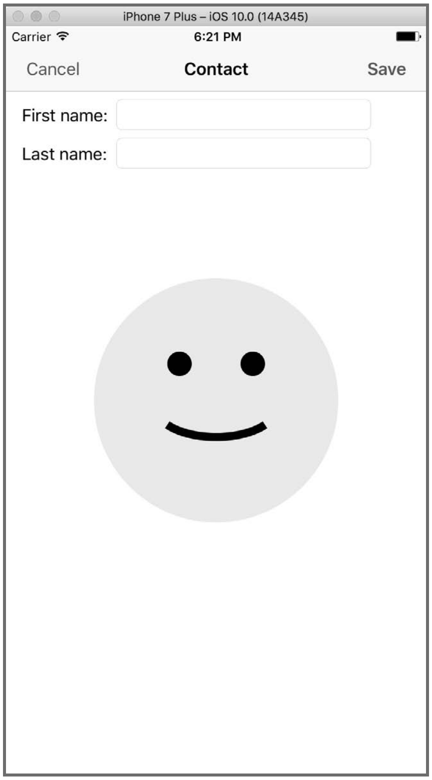


图28-24 新建联系人的默认头像

在本章中，我们开发了一个应用，模拟了现实中常见的编程场景。很多在Swift发布前开发的应用在不远的将来都可能会需要混合语言编程：旧的部分保留Objective-C，新增部分用Swift开发。我们用Objective-C为Contacts写的代码就像是应用“旧的”部分，然后用Swift增加“新的”部分，这样就见识了如何组织需要利用互操作的应用。

给老应用添加Swift代码时，会经常遇到与Cocoa和Foundation框架互操作的情况下。这类互操作是无缝的，系统会自动处理。如你所见，如果是自己所写Objective-C和Swift代码之间的互操作，就会更复杂一些。

28.4　白银挑战练习

给应用增加功能，让用户能够查看联系人信息。用户应该能在`ContactsViewController`中点击一行，然后应用会推一个`UIViewController`到当前的`UINavigationController`栈上。把这个新的视图控制器命名为`ExistingContactViewController`并用Swift实现它。

28.5　黄金挑战练习

用户应该能编辑已有联系人的信息。为`ExistingContactViewController`增加这个功能。确保在这个视图控制器中所做的修改在`ContactsViewController`中可见。

第 29 章 结　语

恭喜，你完成了Swift语言的入门学习！感谢你和我们一起坚持到底。

这一路下来，我们学习了相当多的内容：从Swift的基本特性（比如`let`和`var`）到高级特性（比如泛型和互操作），还有如何把这些知识结合起来写出纯Swift程序，以及把对Swift的理解运用到一些简单的macOS和iOS应用上。现在你已经是一名Swift开发者了。

29.1　接下来学习什么

经过辛苦的学习，你应该对自己的Swift开发生涯抱有什么样的期待呢？事实是你的旅程才刚刚开始。Swift是一门具备丰富特性的语言，我们每天都有大量的机会学到更多的东西。此外，Swift确实开始在和一系列用来开发macOS和iOS应用的苹果框架交互过程中展现出强大的能力。这是你应该聚焦的学习重点。

29.2　插个广告

Matt和John都有Twitter账号，你可以通过@matthewDmathias关注Matt，通过@nerdyjkg关注John。我们在日常发布动图和表情的间隙也会发一些关于Swift编程的有用信息。

如果喜欢本书，请关注https://www.bignerdranch.com/books上的其他Big Nerd Ranch系列图书。我们有Mac和iOS编程的参考资料，也提供为期一周的集训帮助你深入学习这两个平台。访问https://www.bignerdranch.com/training可以获取更多信息。

29.3　邀请

你的Swift知识会随着实践继续增长。花一些时间开始一个新项目，尝试一些新东西。如果手头没有项目，也没什么想法，可以访问https://developer.apple.com，这里很好地概括了对Mac和iOS开发者有用的资料，还有一些可能激发你创意的示例。

另一个建议是找一些本地的Mac和iOS开发者Meetup小组。大多数大城市都有这样的团体，会定期组织讨论。参加这类聚会能帮助你学习、实践以及结识同行。

来加入我们吧。我们在创造，也很乐于看到你能创造些什么。

延 展 阅 读



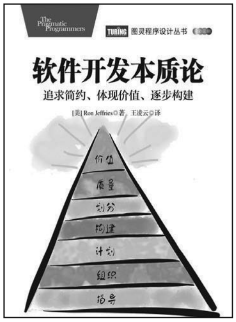
- 中文版累计销量逾70 000册
- 全球数百万开发者交口称赞的iOS开发圣经

第8版即将出版，图灵社区本书主页：http://www.ituring.com.cn/book/1973
作者：Molly Maskrey , Kim Topley , David Mark , Fredrik Olsson , Jeff LaMarche
译者：周庆成



- 你一定能看懂的算法基础书
- 代码示例基于Python
- 400多个示意图，生动介绍算法执行过程
- 展示不同算法在性能方面的优缺点
- 教会你用常见算法解决每天面临的实际编程问题

书号：978-7-115-44763-0
定价：49.00 元



- 敏捷先驱为你直观呈现软件开发简约之道，实践极限编程
- 构建高质量软件系统必读

书号：978-7-115-44110-2
定价：39.00 元



- 技术人员版《人性的弱点》
- 提升职业生涯软技能
- 探讨领导力、合作、沟通、高效等团队成功关键因素
- 2016年最受欢迎电子书 非技术类TOP10

书号：978-7-115-43418-0
定价：45.00 元

- 源自大名鼎鼎的**Big Nerd Ranch**训练营培训讲义，该训练营在Cocoa（之前为AppKit）开发及培训方面有近20年的经验。
- **基础知识**详细介绍+**语言难点**剖析，既适合Swift新手入门，也适合有经验的开发人员深入了解Swift特性。
- 以实际例子阐述知识点，让读者了解**最佳实践**，也让代码可读性更强、更易维护。
- 每章末尾的**“挑战练习”**帮你温故知新，充分巩固学到的知识。

Amazon读者赞誉

“作为一名编程新手，我认为这本书对于理解Swift很有帮助。Big Nerd Ranch用一种易读易懂的方式对Swift进行剖析。拿到这么实用的书很开心，相信我很快就能完成目标，开发出自己的iOS应用。”

“如果你是初学编程，会发现这本书虽然充满挑战，但是简单易懂。如果你有丰富的编程经验，熟悉多门语言，那么这本书能帮助你迅速掌握Swift。”

“这是我目前读过的最佳Swift编程入门书。尽管刚读到一半，但是我学到的知识已经比其他同类图书和在线课程多得多了。真是太棒了！”

“我使用过Objective-C一段时间，编程和苹果平台对我而言都不陌生。我觉得这本书对于学习Swift非常有帮助。作者们不仅详细介绍了Swift的基础知识，并且深入剖析了这门语言深奥难懂的方面。结尾处的代码有关应用开发，但是这本书的关注点主要在于这门语言的特性以及它们为何有用。”

“Swift编程的绝佳入门书，通过诸多例子介绍了如何运用Swift中的特性。不论你是否有编程经验，都需要完成这本书提供的练习，否则不可能做到快速学习。”

Big Nerd Ranch

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/移动开发/Swift

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-45746-2
定价：89.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks